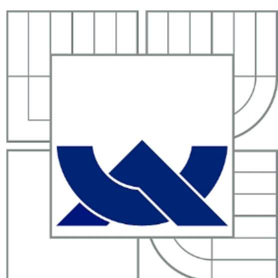
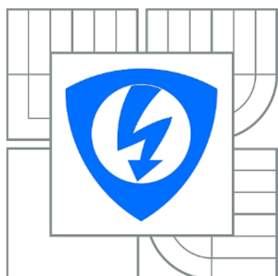


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF TELECOMMUNICATIONS

REALIZACE SUPERPOČÍTAČE POMOCÍ GRAFICKÉ KARTY

REALIZATION OF SUPERCOMPUTER USING GRAPHIC CARD

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE Bc. FILIP JASOVSKÝ

AUTHOR

VEDOUCÍ PRÁCE Ing. JAN MAŠEK

SUPERVISOR

BRNO, 2014

VYSOKÉ UČENÍ
TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

Ústav telekomunikací

Diplomová práce

magisterský studijní obor
Telekomunikační a informační technika

Student: Filip Jasovský **ID:** 125233

Ročník: 2 **Akademický rok:** 2013/2014

NÁZEV TÉMATU:

Realizace superpočítače pomocí grafické karty

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s možnostmi výpočtů na grafických kartách. Dále se seznamte s algoritmy umělé inteligence (např. náhodné lesy). Implementujte algoritmy umělé inteligence vytvořené v prostředí CUDA nebo OpenCL do prostředí jazyka JAVA. Ověřte funkčnost algoritmů na velkých databázích a porovnejte výsledky s algoritmy pro běžné procesory.

DOPORUČENÁ LITERATURA:

[[1] Benedikt Waldvogel. CURFIL – CUDA Random Forest for Image Labeling tasks, 2013.
[2] David B. Kirk and Wen-mei W. Hwu. 2010. Programming Massively Parallel Processors: A Hands-On Approach (1st ed.), San Francisco, CA, USA..

Termín zadání: 10.2.2014 **Termín odevzdání:** 30.5.2014

Vedoucí práce: Ing. Jan Mašek

prof. Ing. Kamil Vrba, CSc.
Předseda oborové rady

ABSTRAKT:

Tato diplomová práce se zabývá realizací superpočítače pomocí grafické karty s použitím technologie CUDA. Teoretická část práce popisuje funkci a možnosti grafických karet a běžných ústrojí stolních počítačů a dějů probíhajících při procesu výpočtů na nich. Praktická část se zabývá vytvořením programu pro výpočty na grafické kartě za použití algoritmu umělé inteligence a to konkrétně umělých neuronových sítí. Následně je vytvořený program použit pro klasifikaci dat z objemného vstupního datového souboru.Na závěr jsou porovnány dosažené výsledky.

KLÍČOVÁ SLOVA:

CPU, GPU, Grafická karta, vyhodnocování dat, CUDA, OpenCL, výpočty, procesory, zrychlení, programování, superpočítač, GPGPU, Neuronové sítě, umělá inteligence

ABSTRACT:

This master´s thesis deals with realization of supercomputer using graphic card with CUDA technology. The theoretical part of this thesis describes the function and the possibility of graphic cards and desktop computers and processes taking place in the proces sof calculations on them. The practical part deals with creation system for calculations on the graphic card using the algorithm of artificial intelligence, more specifically artificial neural networks. Subsequently is the generated program used for data classification of large input data file. Finally the results are compared.

KEYWORDS :

CPU, GPU, Graphic card, data evaluation, CUDA, OpenCL, calculations, processors, acceleration, programming, supercomputer, GPGPU, Neural networks, artificial intelligence

JASOVSKÝ, F. *Realizace superpočítače pomocí grafické karty*.
Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2014. 52 s. Vedoucí diplomové práce: Ing. Jan Mašek.

Prohlášení

Prohlašuji, že svou diplomovou práci na téma Realizace superpočítače pomocí grafické karty jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb..

V Brně dne 30.5.2014 ....

podpis autora

Poděkování

Děkuji vedoucímu diplomové práce Ing. Janu Maškovi za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé diplomové práce.

V Brně dne 30.5.2014 ....

podpis autora

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Technicka 12, CZ-61600 Brno
Czech Republic
http://www.six.feec.vutbr.cz

PODĚKOVÁNÍ

Výzkum popsaný v této diplomové práci byl realizován v laboratořích podpořených z projektu SIX; registrační číslo CZ.1.05/2.1.00/03.0072, operační program Výzkum a vývoj pro inovace.

Brno ....
(podpis autora)

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

OP Výzkum a vývoj
pro inovace

Obsah

ÚVOD **8**

1 VYUŽITÍ GPU PŘI VÝPOČETNÍCH PROCESECH **9**

1.1 Vývoj grafických karet 9

1.2 Rozdíly GPU a CPU 10

1.3 Algoritmy umělé inteligence 14

2 TECHNOLOGIE CUDA A OpenCL **17**

2.1 Technologie CUDA 17

2.2 Standard OpenCL 20

2.3 Nejnovější trendy v paralelní programování na GPU 22

3 IMPLEMENTACE ALGORITMŮ **25**

3.1 Programování CUDA aplikací 25

4 SROVNÁNÍ VÝPOČTŮ GPU A CPU **28**

4.1 Program pro demonstraci výpočtů 28

4.2 Dosažené výsledky 39

ZÁVĚR **47**

Literatura 49

Seznam symbolů a zkratek 51

Obsah přiloženého DVD 52

Seznam obrázků

Obr. 1.1 Porovnání struktury CPU a GPU 11
Obr. 1.2 Architektura SISD 11
Obr. 1.3 Architektura SIMD 11
Obr. 1.4 Architektura MISD 12
Obr. 1.5 Architektura MIMD 12
Obr. 1.6 Architektura SIMT 13
Obr. 1.7 Sdílená, distribuovaná a kombinovaná paměť 13
Obr. 1.8 Znázornění umělého neuronu 15
Obr. 2.1 Architektura technologie CUDA 17
Obr. 2.2 Dílčí části technologie CUDA 19
Obr. 3.1 Ukázka volání kernelu 25
Obr. 4.1. Propojení jazyků Java a C++ 28
Obr. 4.2 Ukázka vytvořeného kódu pro načtení datového souboru 29
Obr. 4.3 Ukázka zápisu kódu pro předávání dat C++ 30
Obr. 4.4 Hlavičkový soubor vytvořený pomocí javah 31
Obr. 4.5 Cykly výpočtu na CPU 34
Obr. 4.6 Paralelizace 1 na GPU 34
Obr. 4.7 Ukázka kódu paralelizace 1 35
Obr. 4.8 Znázornění paralelizace 2 36
Obr. 4.9 Ukázka funkce synchronizace vláken 36
Obr. 4.10 Ukázka principu pararedukce 37
Obr. 4.11 Výpis z programu 38
Obr. 4.12 Parametry použité grafické karty 39

Seznam tabulek

Tab. 4.1 Porovnání výsledků CPU a GPU 40
Tab. 4.2 Výsledky správné klasifikace vstupních dat 44

Seznam grafů

Graf 4.1 Závislost hodnot výpočtu na době trvání výpočtů CPU a GPU 42
Graf 4.2 Závislost urychlení výpočtů pomocí GPU na počtu řádků pro klasifikaci 43
Graf 4.3 Závislost úspěšnosti klasifikace na použitém počtu řádků pro trénování 45

ÚVOD

Vývoj informačních technologií jde stále nezadržitelně kupředu a díky tomu se nám otvírají stále nové a lepší možnosti využití nových systémů a struktur. V minulosti se málokteré zařízení dalo z dnešního pohledu označit za rychlé nebo výkonné. Dnešní počítače a všechny jejich variace mají často podstatně vyšší výkon jednotlivých komponent, než je pro jejich běžné používání vůbec nutné. Mezi takovéto komponenty dnes patří grafické karty.

Jejich vývoj byl dlouhý a velice pestrý a jejich použití se s navyšujícím výkonem také měnilo. Dnes už se výkon grafických karet dostal na úroveň počítačových sestav a tak nám je umožněno za pomocí některých technologií využít jejich vysoký výkon nejen pro práci s grafikou ale také jako další výpočetní systém v jednom počítači. Můžeme tak výrazně snížit zatížení procesoru a operačních pamětí počítače a nechat námi požadované výpočetní operace provést samotnou grafickou kartu.

Hlavní přínos této práce spočívá v implementaci algoritmu umělé inteligence, vytvořeného autorem této diplomové práce, za účelem demonstrace výpočetního výkonu grafických procesorů osazených na grafické kartě při výpočtech nad velkým množstvím vstupních dat ve srovnání s výpočty prováděnými nad stejnými daty za pomoci CPU. Jedná se zde konkrétně o algoritmus umělých neuronových sítí, který je po vytvoření vhodného programového kódu určeného k paralelizaci na GPU schopen provádět výpočty na grafické kartě a tím až 17x urychlit celý proces v porovnání s použitím klasických výpočtů za pomoci klasického procesoru osazeného na základní desce běžného počítače. K tomuto účelu poslouží technologie CUDA nebo OpenCL.

Zbytek této práce je členěn následovně. V první kapitole jsou shrnuty informace o moderních grafických kartách, jejich vývoji, rozdíly v porovnání s procesory stolních počítačů a známé algoritmy umělé inteligence. Ve druhé kapitole jsou následně popsány technologie CUDA a OpenCL a nejnovější trendy této oblasti. Třetí kapitola pojednává o implementaci algoritmů pro výpočty na GPU a použití technologie CUDA při programování. Ve čtvrté kapitole je popsána praktická část práce, jsou zde shrnuty výsledky implementace algoritmu a jejich srovnání.

1 VYUŽITÍ GPU PŘI VÝPOČETNÍCH PROCESECH

V této kapitole je popsán vývoj grafických karet a s ním spojené změny práce s procesory, vlákny a paměťmi. Dále jsou zde uvedeny informace o rozdílech mezi GPU a CPU při práci s instrukcemi a daty. Na závěr této kapitoly jsou popsány některé algoritmy umělé inteligence.

1.1 Vývoj grafických karet

Není tomu ještě tak dávno, kdy téměř všechny existující programy byly pouze jednovláknové (tzv. single-thread), což znamená, že je procesor počítače (CPU – central procesor unit) vykonává bez možností rozložení výpočtů do více vláken najednou. Postupem času se začaly objevovat více jádrové procesory a s nimi i více vláknové programy (tzv. multi-thread), které umožňovaly rozložení výpočetního výkonu dovíce vláken a díky tomu výrazné urychlení vykonání celého programu. Ale i ty nejlepší a nejvíce jádrové procesory, které lze dnes běžně koupit, mají jen velice málo vláken na čtyř-jádrovém procesoru je to 8 vláken. Pokud je tedy daný program vytvořen pro použití s více vlákny současně, umožní nám to na vhodném procesoru provést výpočet podstatně rychleji, než kdybychom tento program pouštěli v jednom vlákně. Mezi základní funkce CPU patří obsluha všech žádostí, které se během provozu počítače neustále objevují, takže neslouží čistě jen pro výpočty. [17]

Grafické karty jsou zcela jiným případem. Mají totiž řádově desítky až stovky tzv. Stream procesorů, které jsou schopny samostatně a velice rychle provádět výpočty. Pomocí technologie CUDA lze z kteréhokoli jedno-vláknového programu udělat více-vláknový a provést výpočty na desítkách či stovkách stream procesorech, což vede k vytvoření až tisíce vláken současně a tak docílit propastně většího výpočetního výkonu než s použitím CPU.

Na první pohled se může zdát, že při přihlédnutí ke všem skutečnostem by nám teoreticky měla grafická karta posloužit jako opravdu velice výkonný počítač. Opak je ale pravdou. Grafické karty jsou sice schopny opravdu obrovského výpočetního výkonu při zpracování dat, ale nejsou schopné provádět standardní obslužné operace, jak tomu je u CPU počítače. Proto celý systém využití grafických karet při složitých

výpočtech funguje tak, že vše kromě samotných výpočtů provádí CPU a GPU si „zavolá na pomoc“ až při výpočtech. Tento fakt ale není ničemu na škodu, protože díky tomu není zatěžován CPU a počítač může být opravdu rychlý.

Kvůli stále většímu využívání grafických karet jako výpočetních jednotek například v serverech byly vytvořeny společností NVidia karty s názvem TESLA, které nedisponují žádným grafickým rozhraním integrovaným přímo na kartě. Tyto karty jsou využity pouze pro paralelní výpočty a dosahují obrovských výpočetních výkonů. Například grafická karta s označením TESLA K40 obsahuje čip s 2880 stream procesory. Pokud se v budoucnu bude na vývoji výpočtů na GPU dále intenzivně pracovat, mohl by se pak obyčejný počítač osazený procesorem o nikterak vysokém výkonu a jednou nebo dvěma grafickými kartami s vysokým výkonem dostat na úroveň dnešních tzv. superpočítačů.[2]

Toto všechno je velmi dobrou realizací myšlenky, jak velice efektivně využít všechny prostředky počítače jako celku, kdy se využije výkonu, který teoreticky počítač má, ale prakticky ho nevyužívá.

1.2 Rozdíly GPU a CPU

V dnešní době existují dva směry vývoje procesorů a to vícejádrové (multi-core) procesory a mnohojádrové (many-core). První kategorii můžeme najít u klasických procesorů, kde procesor obsahuje několik jader, které jsou svoji funkčností naprosto plnohodnotné, mají různé logické obvody, paměť cache, zajišťují zpracovávaní instrukcí apod. Do druhé kategorie patří dnes grafické procesory, které mohou obsahovat řádově desítky, stovky až tisíce jader, která jsou jako zjednodušená doplněná o řídící logické obvody a mají případně i malou cache paměť. [18]

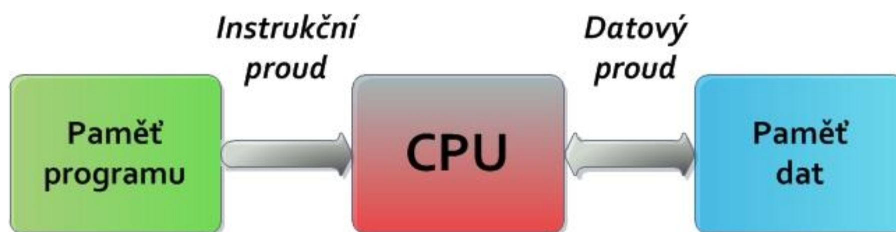
Shlukům výpočetních jader říkáme multiprocesory a pomocí paměti, která je sdílená slouží pro výměnu dat. Vzhledem k počtu procesorů může současně běžet až tisíce různých vláken, které jsou přepínané, aby nedocházelo k zatěžování sdílené paměti.

Vzhledem k odlišné vnitřní struktuře obou kategorií procesorů se i jinak provádí jejich programování. Na obrázku 1.1 můžeme vidět přibližné porovnání klasického dnešního čtyř-jádrového procesoru (vlevo) a mnoha procesorové grafické karty (vpravo). Z obrázku je patrné, že klasický procesor obsahuje mnoho dalších částí než jen výpočetní a naopak na grafické karty najdeme převážně jen procesory pro výpočty.

Obr. 1.1 Porovnání struktury CPU a GPU

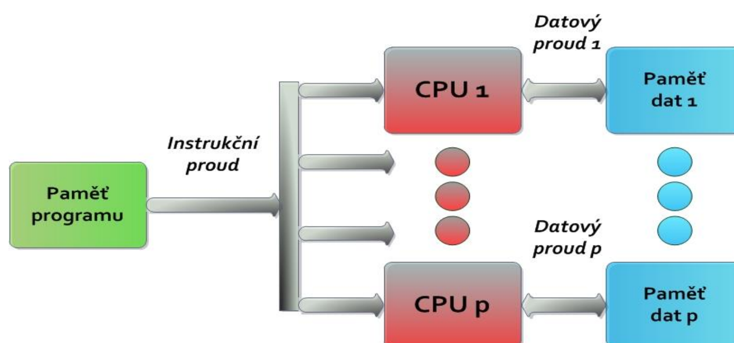
Zpracování instrukcí, jejich počet a datové proudy jsou kategorie, kterým se zabývá Flynnova taxonomie paralelních architektur. Jeho rozdělení je popsáno níže.

SISD – Jeden instrukční a jeden datový proud (Single Instruction Single Data stream) je kategorie ukázaná na obrázku 1.2. Z paměti programu jde jeden instrukční proud do jediného CPU a z něho jeden datový proud do jediné paměti dat a zpět.



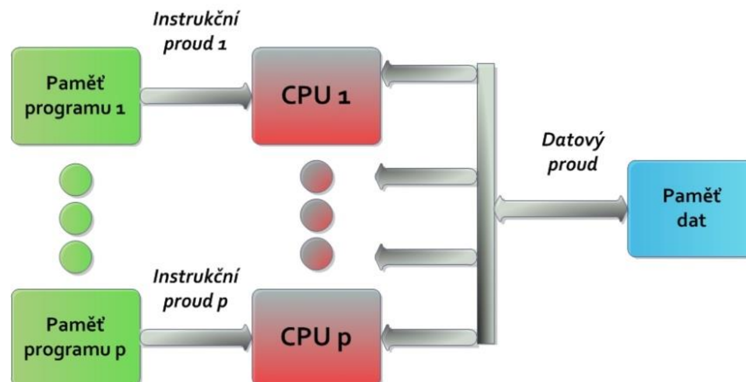
Obr. 1.2 Architektura SISD

SIMD – Jeden instrukční proud a více datových proudů (Single Instruction Multiple Data stream). Z paměti programu vede jeden instrukční proud do více CPU a z nich jde stejný počet datových proudů do pamětí a zpět.



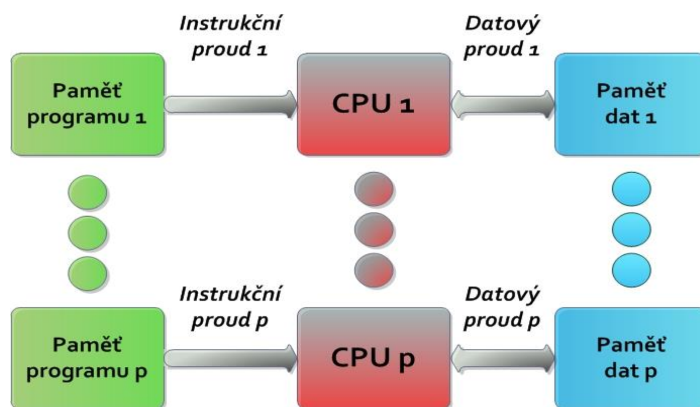
Obr. 1.3 Architektura SIMD

MISD – Více instrukčních proudů a jeden datový proud (Multiple Instruction Single Data stream). Několik instrukčních proudů vede ze stejného počtu pamětí programu do několika CPU a odtud pomocí jediného datového proudu do paměti dat a zpět. Tato architektura je uvedena na obrázku 1.4. [18]



Obr. 1.4 Architektura MISD

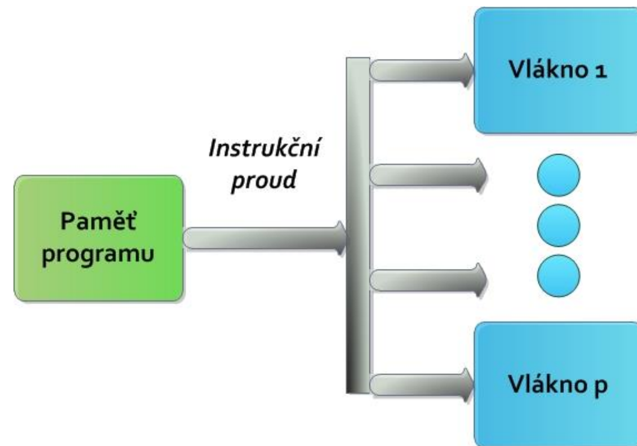
MIMD – Více instrukcí a více datových proudů (Multiple Instruction Multiple data stream). Několik instrukčních proudů směřuje ze stejného počtu pamětí programu do CPU jednotek a odtud vede několik datových proudů do pamětí dat a zpět. Tato architektura se používá například u vícejádrových procesorů, kdy procesory provádí různé instrukce a pracují s různými daty.



Obr. 1.5 Architektura MIMD

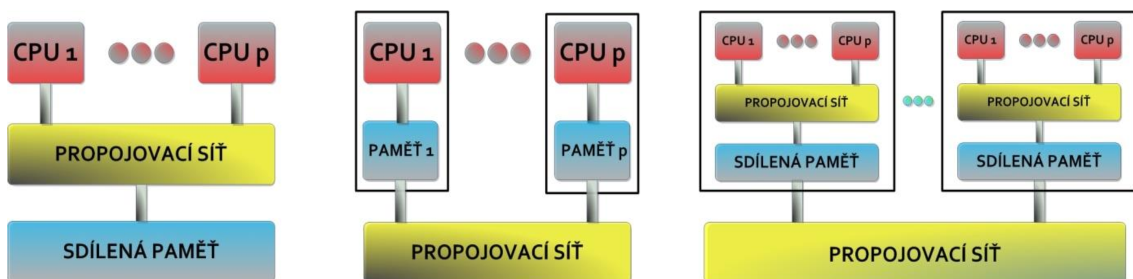
SIMT - Jedna instrukce pro více vláken (Single Instruction Multiple Threads) je architektura používaná při výpočtech na grafických kartách. Vstupní instrukce se rozdělí mezi potřebný počet vláken, která tuto instrukci vypočítávají paralelně. Tato architektura je podobná se SIMD, ale počet vláken se zde může pohybovat v řadově stovkách až tisících. Použití této architektury napomáhá velkému výpočetnímu výkonu grafických karet. Karty jsou díky možnosti použití velkého množství vláken

a s tím spojeným efektivním systémem paralelních výpočtů velice vhodné pro složité výpočty velkého objemu dat a to především tam, kde se výpočty provádí v dlouho trvajících cyklech apod. [17]



Obr. 1.6 Architektura SIMT

Je zde použito také několik typů pamětí, jako je paměť sdílená, paměť distribuovaná a kombinace těchto dvou pamětí. Pokud je použita paměť sdílená, tak do ní přistupují všechny procesory, pokud je použita distribuovaná, tak má každý procesor svoji paměť a v případě že je nutný přístup do paměti jiného procesoru, provede se tento přístup přes propojovací síť. Na obrázku 1.7 je znázorněn systém se sdílenou pamětí (vlevo), systém s distribuovanou pamětí (uprostřed) a kombinovaný systém se sdílenou i distribuovanou pamětí (vpravo). [17]



Obr. 1.7 Sdílená, distribuovaná a kombinovaná paměť

1.3 Algoritmy umělé inteligence

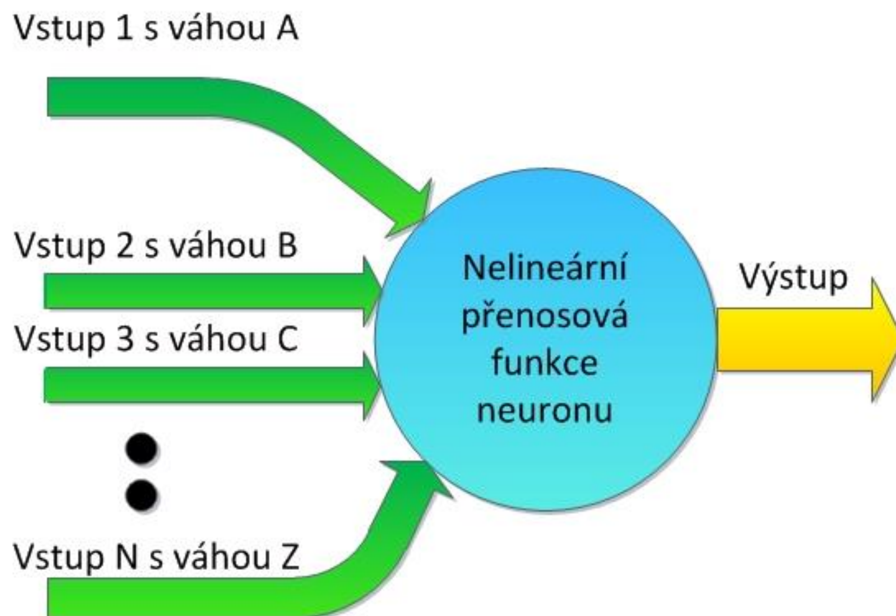
Pokud je požadavek na to, aby se program sám rozhodoval a učil se, je možné použít některý z učících se algoritmů, které patří do skupiny algoritmů umělé inteligence.

Neuronové sítě - jsou inspirovány funkcí lidského mozku, kde je velké množství buněk zvaných neurony, které jsou schopny vzájemně komunikovat pomocí slabých elektrických impulzů. Umělá inteligence je často inspirovaná člověkem samotným, přesněji jeho mozkem a propojením všech různých částí lidského těla s řídícím prvkem, kterým právě lidský mozek je. U neuronových sítí umělé inteligence je použit jako základní prvek umělý neuron, který je propojen s dalšími a to vždy tak, že jeden neuron může mít libovolný počet vstupů, ale má vždy pouze jeden výstup. Neurony si mezi sebou předávají informace a zároveň na ně aplikují určité přenosové funkce, a tím mění obsah předávaných informací. [5]

Mezi nejdůležitější základní požadavky na umělou inteligenci patří schopnost se učit, čehož jsou právě neuronové sítě schopny. Pokud necháme tento algoritmus zpracovat tzv. trénovací data, tak je schopen si zapamatovat, na základě kterých kombinací uvnitř sítě bylo dosaženo požadovaného výsledku. S příchodem jiných, podobných dat, je potom síť schopna vyhodnotit tyto data a určit, kterým výsledkům trénovacích dat jsou aktuální výsledky nejvíce podobné. Jako příklad si představme několik aut, která mají některou ze základních barev jako je červená, zelená, modrá, bílá a černá. Pokud naučíme pozorovatele (neuronovou sít) která barva je která, bude schopen po projetí auta například růžové barvy určit, které barvě doposud naučené, se tato barva nejvíce podobala, takže by v našem případě určil, že se jednalo s největší pravděpodobností o auto s barvou blížící se červené. Je zde tedy docíleno toho, že bez předchozí znalosti růžové barvy byl pozorovatel schopen určit, že nejpodobnější barvou k růžové je z palety jeho naučených barev červená.[20]

S využitím neuronových sítí je možné se setkat i u řešení nelineárních úloh. Pokud nastane případ, že díky pestrosti vlivů, které ovlivňují variabilitu sledované proměnné, není možné aplikovat základní matematickou funkci, nabízí se použití neuronových sítí, které jsou částečně schopny pracovat i s šumy a nepřesnými daty. [3]

Na obrázku 1.8 je znázorněn umělý neuron s několika vstupy a jedním výstupem. Vstupy mohou mít určitou váhu, která se během fáze učení a i později při zpracování dat může měnit. Samotný neuron obsahuje nějakou nelineární přenosovou funkci, kterou upravuje tok informací.



Obr. 1.8 Znázornění umělého neuronu

Náhodné lesy – jedná se o model složený z několika stromů, které nemusí být pouze binární. Tato skupina stromů následně rozhoduje o zařazení objektu do tříd a to za pomoci jejich klasifikačních funkcí, které by měly být vhodně zkombinované.

Bagging je první základní metodou pro generaci klasifikačních stromů. Jeho princip spočívá ve vytváření určitého počtu souborů, kde jejich počet je označen jako *k*. Tyto soubory se vytváří náhodným výběrem ze souboru určeného k trénování. Na vytvoření klasifikačního stromu se následně použije každý soubor z tohoto výběru. Pomocí většinového hlasování za použití totožných vah následně dojde k vytvoření klasifikačního lesu. [8]

Další metodou je Boosting. Tento název je odvozen od anglického slova boost, což v překladu znamená zesílení. U této metody se klasifikační strom vytváří za pomoci snižování a zvyšování vah jednotlivých vstupů. Ke zvyšování váhy dochází u případů, kde došlo ke špatné klasifikaci. Naopak u případů, kde proběhla klasifikace správně se váhy zmenšují. Pro toto přiřazování větší a menší váhy jsou použita trénovací data. Tento postup se zaměřuje na takové případy, kdy nedochází ke korektnímu řazení do adekvátních tříd. [8]

U metody náhodných lesů není na rozdíl od předchozích dvou metod snaha o dosažení stromů s co největší přesností. Nezáleží zde tedy u jednotlivých stromů na jejich kvalitě. Cílem je minimalizace chyby celého lesa a ne jednotlivých dílčích stromů. Pro nalezení nejvhodnějšího štěpení u jednotlivých uzlů se používají tzv. prediktory. Výsledný počet prediktorů je vybrán z jejich celkového množství a to jako jejich náhodná podmnožina. Stromy u této metody mohou růst až do největší

velikosti a to bez potřeby je jakkoli upravovat. Nejdůležitějšími parametry metody náhodných lesů je celkový počet použitých stromů a také počet proměnných, které jsou náhodně vybrány a díky kterým dochází k rozdělení uzlů. [8]

2 TECHNOLOGIE CUDA A OpenCL

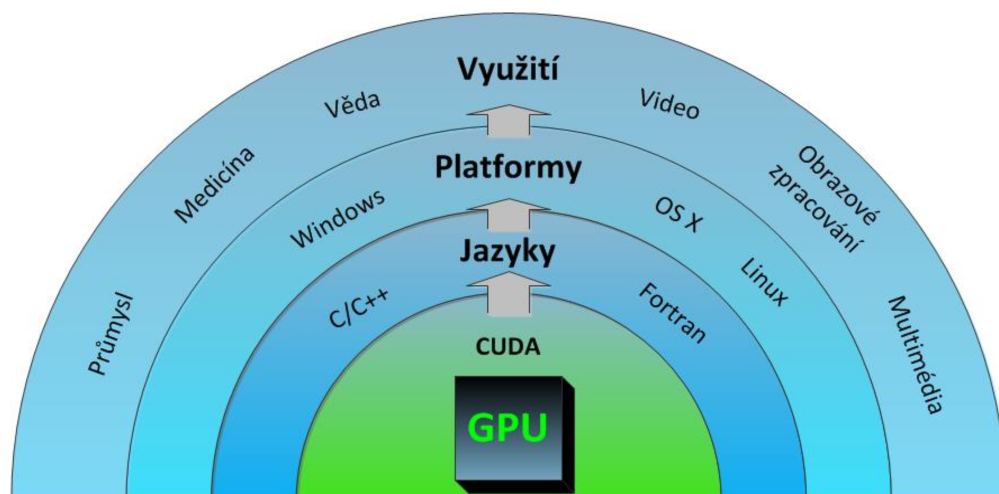
Tato kapitola pojednává o technologii CUDA standardu OpenCL. Jsou zde uvedeny základní informace o jejich částech a funkci.

Společnost Nvidia vyvinula a v roce 2007 zveřejnila technologii pro paralelní výpočty na GPU grafických karet s názvem CUDA (Compute Unified Device Architecture), přičemž první verze byla číslo 1.0, půl roku na to vyšla verze 1.1 a takto se postupně objevovaly stále nové a vždy v něčem jiné verze 2.x, 3.x, 4.x až po poslední verzi 6.0. Alternativou je použití technologie OpenCL (Open Computing Language) patřící pod Khronos Group, do které patří množství velkých společností jako je AMD, Intel, IBM, Nvidia apod.

Díky těmto nástrojům lze vytvářet a spouštět programy na procesoru grafické karty GPU (Graphic Processing Unit), používat paměti karty pro mezivýpočty a ukládání dočasných dat apod. Programy spustitelné na grafické kartě se píší převážně v programovacím jazyku C++, případně FORTRAN.

2.1 Technologie CUDA

Technologie CUDA je spustitelná na většině novějších grafických karet společnosti Nvidia a to jak pod operačním systémem Windows, Mac OS X i Linux, její kompatibilitu s konkrétní kartou můžeme ověřit na webových stránkách společnosti Nvidia. Existuje i emulátor grafické karty Nvidia, pomocí kterého lze spustit technologii CUDA i na běžném CPU.



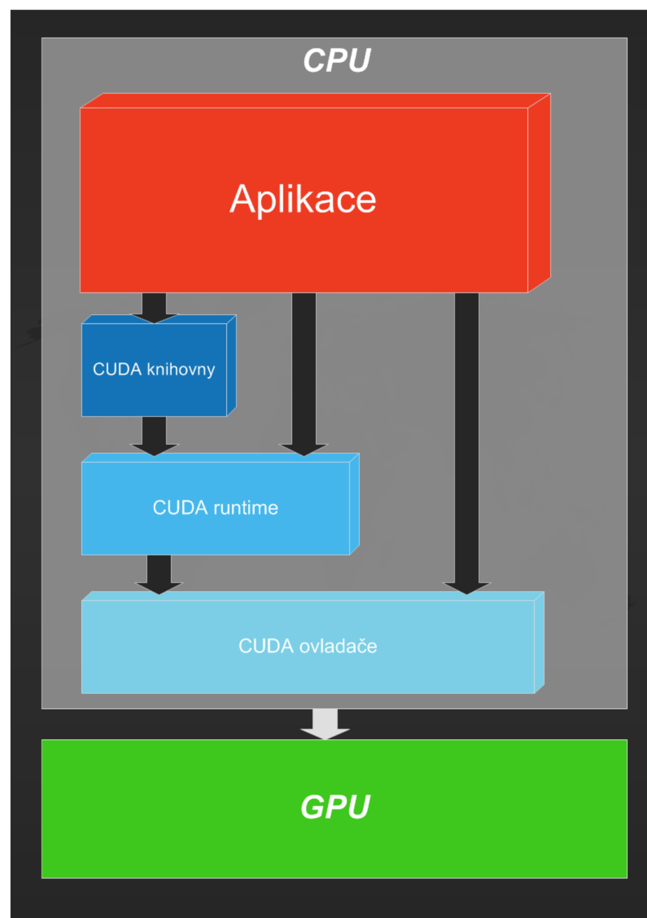
Obr. 2.1 Architektura technologie CUDA

C++ thrust API - jedná se o vysokoúrovňové rozhraní, které disponuje sadou datově paralelních jednoduchých funkcí, jako jsou například řazení a redukce. Díky těmto funkcím je výrazně usnadněn vývoj aplikací a čitelnost spolu s přehledností zdrojových kódů. Programování je snadnější protože API dokáže řešit mnoho věcí automaticky, například konfiguraci kernelů, které jsou spuštěné. Je zde předpoklad, že vzhledem k jeho optimalizaci budou schopny příští verze běžet rychleji a to bez nutnosti zásahu do zdrojových kódů. [3]

CUDA runtime API- nabízí jednodušší kontrolu nad kódem, chybami a přístupem k programovým možnostem týkajících se hardwaru a to za pomocí jen pár syntaktických rozšíření v jazyce C. Musí být použit zvláštní překladač NVCC, aby došlo ke správnému překladu. Tento překladač slouží pro oddělení kódů pro GPU a CPU. [2]

CUDA driver API - slouží pro nejvyšší kontrolu nad kódy aplikací, které to vyžadují. Toto API je nízkoúrovňové a oproti runtime API zvládá vytváření kontextů, zapouzdření dat a kódů při práci s moduly a kvalitnější dohled nad spouštěním jednotlivým kernelů a přenosem dat. Nevýhodou je složitost ladění a programování aplikací, neboť je zapotřebí více kódu, ale na druhou stranu je velkou výhodou možnost použití kteréhokoliv překladače v jazyce C. [17]

Aplikace, která je spuštěná může v kterékoli chvíli přistupovat ke všem částem API CUDA. Na obrázku 2.2 jsou zobrazeny dílčí části technologie CUDA ve vztahu k aplikaci a jejich vzájemné propojení a následné propojení se samotným GPU.



Obr. 2.2 Dílčí části technologie CUDA

Grafické procesory dnešní doby jsou vyladěny tak, aby dokázaly řešit úlohy, ve kterých je spouštěna jedna a tatáž funkce nad určitou množinou nezávislých prvků datového charakteru, což se nazývá datově paralelních úloh.

Základním prvkem u programování CUDA architektury je kernel, což je vlastně část kódu, u kterého je požadavek na spuštění paralelně nad elementy datového typu. V jazyce C je kernel definován, jako funkce spouštějící se na straně CPU. V prostředí CUDA je tato strana nazývána jako hostitel (host), avšak její provedení je vykonáváno na grafickém procesoru, který se označuje jako zařízení (device). Vlákna (threads) jsou vytvářena na zařízení z důvodu spuštění kernelu. [3]

Tyto vlákna se organizují do tzv. bloků a bloky následně do mřížky, přičemž velikost bloku a počet bloků je volen programátorem a to při spouštění kernelu. Spouštění kernelu je prováděno asynchronně, což má za následek, že na straně hostitele se nečeká na dokončení výpočtu prováděného na zařízení, ale dále se pokračuje ve vykonávání kódu sekvenčního charakteru. Za předpokladu, že vzhledem k fyzickému oddělení hostitele a zařízení, má každý z nich svůj adresný

prostor, lze docílit překrývání prováděných výpočtů na zařízení a hostiteli. Spojením CPU a GPU vznikne heterogenní systém. Tento systém dokáže vytvořit velice efektivní kombinace paralelních výpočtů, které potřebují mnoho operací s plovoucí řádovou čárkou a klasických aplikací, které jsou vícevláknové a běží na vícejádrových procesorech.[10]

Mřížky i bloky mohou být jednorozměrné, dvourozměrné i třírozměrné. V rámci bloku threat IDX má každé vlákno svůj identifikátor. Tento identifikátor je vždy jedinečný a stejně tak má každý blok v rámci mřížky svůj index block IDX. Tyto indexy nám slouží k rozlišení vláken a na základě nich si musí samo vlákno rozhodnout o přístupu k datům, které bude používat.[7]

Pokud potřebujeme zpracovávat objemná data nebo obrázky, použijeme identifikátory třírozměrného typu, které jsou zavedeny pro zjednodušení indexace dat u řešení vícerozměrných problémů. Vlákna je možné pomocí bariéry vzájemně synchronizovat a zároveň může docházet v rámci bloků k vyměňování informací skrze sdílenou paměť. [13]

Pro paralelní nebo sériové spouštění musí být mezi sebou bloky vláken vzájemně nezávislé. Tato nezávislost vede k možnosti libovolného rozmístění jednotlivých bloků mezi multiprocesory a to umožňuje programátorům vývoj takových aplikací, které mají vzhledem k počtu multiprocesorů dobrou škálovatelnost.

Data, nad kterými pracují kernely, obsahuje globální paměť grafické karty a kvůli tomu musí být do zařízení nakopírována všechna potřebná data, se kterými bude pracovat kernel, a to před jeho samotným spuštěním. Vzhledem k tomu, že kopírování dat je obecně zdlouhavá operace, je tedy vhodné udržovat data nacházející se v globální paměti co nejdéle, tedy bez kopírování do paměti hostitele. [18]

Aby došlo k plnému vytížení zařízení, je nutné spustit potřebné množství vláken. Tím je možné odstranit latence, které vznikají při přístupu do pomalé globální paměti. Pokud k jejich odstranění nedojde, tak se velmi rychle sníží výsledná efektivita jednoho kódu.

2.2 Standard OpenCL

Použití standardu OpenCL není omezeno na grafické karty konkrétního výrobce, jak tomu je u technologie CUDA, na druhou stranu má složitější rozhraní.

To, jak se OpenCL chová je popsáno čtyřmi základními modely. Prvním je model platformy. Hostitel (Host) umožňuje běh OpenCL aplikací a jedná

se o podpůrný systém, může zahrnovat více zařízení jako je CPU, GPU. Zařízení (OpenCL device) udává, kde bude OpenCL aplikace spuštěna. Každé zařízení je složeno z výpočetních jednotek CU (Compute Units). Jsou zde také výpočetní elementy PE (processing elements), ze kterých se skládají výpočetní jednotky. [9]

Dále je zde model prováděcí (execution), který zahrnuje spouštění kernelu a definici kontextu. V OpenCL dochází u programu k rozdělení na dva soubory. První soubor, ve kterém dochází k inicializaci prostředků pro cílové zařízení, na kterém se budou provádět výpočty, se nazývá inicializační a spouštěcí. Druhý soubor obsahuje kernely a funkce pro OpenCL, kde je kernel funkcí jazyka C a vždy je pro něj použito klíčové slovo `__kernel`. Spouštění kernelů samotných se řídí za pomoci příkazů front (command-queue), kde se jedná o objekty pro uložení příkazů OpenCL. S použitím několika těchto front je možné provádět paralelně několik příkazů a to bez nutnosti synchronizace.

Programový (programming) model je třetím základním modelem standardu OpenCL. Vykonávání kernelu je dáno funkcí vícedimenzionálních indexů (NDRange). Instance kernelu nazýváme pracovními jednotkami (WI). V rámci indexového prostoru NDRange tvoří vždy jedna instance jeden prvek. Souhrn alokovaných WI vytváří následně globální prostor, jenž se dělí na pracovní skupiny WG, které mají stejnou velikost. Sdílení dat za pomoci globální nebo lokální sdílené paměti může být prováděno uvnitř každé skupiny pracovními jednotkami. [9]

Paměťový model je posledním základním modelem OpenCL. Tento model obsahuje několik typů paměti, jenž jsou od sebe odlišeny dobou přístupu, způsobem jejich využití a právy, jimiž disponují. Prvním typem paměti je globální paměť (Global Memory). Jedná se o paměť hlavní, která umožňuje jak zápis, tak čtení pro všechny pracovní jednotky. Druhá je paměť pro konstanty (Constant Memory). Jde o oblast paměti globální, ve které zůstává její obsah konstantní a to během spuštění kernelu. Slouží pro uložení dat programátorem a pracovní jednotky mají možnost z této paměti pouze číst. [18]

Dalším typem je paměť lokální (Local Memory), která slouží pro sdílení dat pracovních jednotek uvnitř konkrétní pracovní skupiny. Díky této paměti dochází ke snížení intenzity přístupu do paměti globální. Práva na zápis a čtení v rámci této paměti mají pouze pracovní jednotky. [9]

Soukromá paměť (Private Memory) je typem privátní paměti, z čehož vyplývá, že každá pracovní jednotka má v této paměti vyhrazené svoje místo a pouze do tohoto místa paměti může přistupovat. V rámci standartu OpenCL se jedná o nejrychlejší paměť. U této paměti není potřebná synchronizace. Pokud jsou data příliš velká a do soukromé paměti se nevejdou, může pro jejich uložení posloužit i globální paměť.

Posledním typem paměti je paměť hostitele (Host Memory), jenž slouží pro uložení dat programu. K této paměti má přístup pouze programátor.

2.3 Nejnovější trendy v paralelní programování na GPU

Využívání grafických karet pro výpočty je stále častější i v různých vědeckých a technických odvětvích. Jejich velké využití je možno nalézt například v biologii, kde v posledních letech exponenciálně roste množství dostupných biologických dat, díky čemuž roste i složitost výpočtů při jejich analýze. Při použití klasických procesorů pro výpočty se tyto procesory rychle dostávají na svůj výpočetní limit a navíc nejsou určeny pro výpočty mnoha vláknových aplikací. Grafické procesory jsou na rozdíl od těch klasických cenově dostupnější, nabízí vysoký výpočetní výkon, kterému klasické CPU nedokážou konkurovat a vzhledem k jejich velice rychlému vývoji stalé roste jejich maximální výpočetní výkon. Díky více-jádrové a více-vláknové architektuře mohou být biologická data vypočítávána současně, kdy se nad podskupinami dat provádí současně tisíce výpočtů. V poslední době na GPU přesunuto veliké množství bioinformatických výzkumů, nástrojů a algoritmů, které byly navrženy pro co nejefektivnější využití potenciálu grafických jader. Očekává se tedy, že s rychlým vývojem grafických karet a zvyšujícím se množstvím biologických dat, která je potřeba zpracovat, bude použití GPU při těchto výpočtech stále častější. Pravděpodobnost, že by klasické CPU dohnaly ve své výkonnosti množství grafických procesoru, je velice malá a vzhledem k architektuře obou systémů téměř nereálná.[11]

K využití možností výpočtu na GPU přistoupili také vývojáři ze společnosti Eutechnyx, zabývající se vývojem automobilových závodních her. Ve své novince s názvem NASCAR 2014 se vývojáři zaměřili na co nejdetailnější hrací prostředí a především na efekty během hry. Právě například kouřové efekty jsou téměř vždy tím nejnáročnějším, co grafické karty musí u takovýchto her vypočítávat. Za pomoci technologie CUDA zde došlo ke spuštění stávajícího jednoho vlákna na CPU, jehož výpočet byl urychlen za pomocí stovek GPU jader a to s použitím nejnovějších a nejmodernějších grafických karet GeForce GTX společnosti NVidia. Díky tomu bylo možné zvýšit objem kouře při různých herních situacích až o desetinásobek.

Stále rozšiřující se oblastí pro využití paralelních výpočetních architektur je také medicína. Vývoj nejnovějších medicínských zařízení pro přesné analýzy a výzkumy jde rychle dopředu a proto je potřebné zabezpečit také adekvátní výpočetní výkon při následném zpracování dat z těchto přístrojů. Například některé nemoci je potřeba identifikovat ve velice krátkém čase, kde se může jednat i důležité hodiny.

S použitím vysokého výpočetního výkonu grafických karet pro velký objem zpracovávaných dat lze dosáhnout výrazného zrychlení o proti standardním výpočtům na CPU a to především díky možností paralelizace a velkému počtu grafických jader, jejichž hodnota závisí na použité grafické kartě.[14]

Kryptovací algoritmy a hešovací funkce patří mezi oblasti, kde se začíná využívat možnosti použití paralelních výpočtů za pomocí grafických karet. Pokud jejich tvůrci potřebují zkusit odolnost těchto algoritmů, pokouší se v rámci testování sami o jejich prolomení.To jim pomůže k zjištění, jak moc jsou nebo nejsou tyto algoritmy odolné vůči potenciálním útokům třetích stran. Kryptografické algoritmy se používají pro zabezpečení dat přenášených například po síti, umožňují dohledat autora šifrovaných dat apod. Hešovací funkcí rozumíme takovou matematickou funkci, která slouží k vypočítání kontrolního součtu. Takováto funkce musí být po kryptografické stránce bezpečná. Nehledě na velikost vstupů má vždy stejnou délku. Prolomení takových bezpečnostních algoritmů je velice náročné na výpočetní výkon a s tím i spojený čas potřebný k prolomení takového kódu. Jedinou možností jak takovýto šifrovací algoritmus prolomit je zkoušet všechny možné kombinace znaků.[15]

Počet kombinací záleží na počtu použitých bitů bezpečnostního kódu. Pokud se použije dostatečně bitově dlouhý bezpečnostní součet, může jeho prolomení trvat na běžně výkonném počítači desítky, stovky až tisíce let.To záleží také na použití znaků, tedy zda se používají pouze číslice, malá písmena a čísla nebo kombinace čísel a malých a velkých písmen. Při použití paralelismu na grafické kartě je možno rozdělit počítaní kombinací do tisíců vláken, která zkouší různé kombinace současně. Díky tomu je možné docílit s použitím přesunutí výpočtů na GPU urychlení prolamování těchto bezpečnostních algoritmů řádově až v desetinásobcích času potřebného na CPU. S použitím více grafických karet současně se může jednat až o stonásobky zrychlení. Vývojáři zabezpečovacích algoritmů musí tedy myslet na fakt, že se stále se zvyšujícím potenciálním výkonem nově vyvíjených grafických karet, bude třeba tyto algoritmy vytvářet stále komplikovanější a proti neoprávněnému prolomení odolnější. Pokud tak neučiní, snadno se v budoucnu pak může stát, že algoritmy vyžadující dle původních odhadů a propočtů až stovky let, bude schopna skupina grafických karet prolomit za několik týdnů či dní. [12]

Zabezpečovací algoritmy se dnes používají u většiny autentizací například při přístupu do internetového bankovnictví, přihlašování se do emailových účtů nebo také do dnes již tolik oblíbených sociálních sítí. Je tedy velice důležité, aby zůstali uživatelé těchto služeb v bezpečí po stránce internetové kriminality. Z jiného pohledu na věc je tento příklad vhodnou ukázkou obrovského výkonového potenciálu GPU, který do budoucna má. [21]

Výpočty složitých diferenciálních rovnic a jejich zpracování za pomoci výpočetní techniky je také oblast, kde se začínají uplatňovat paralelní výpočty prováděné na grafických procesorech. Tyto rovnice slouží například pro modelování systémů, které zahrnují přenos tepla, elektrostatiky či akustiky. Potřeba stále složitějších modelů zvyšuje náročnost na jejich zpracování. Protože řešení těchto typů rovnic je neodmyslitelně paralelní, nabízí výpočty prováděné na GPU atraktivní řešení, které výrazně snižuje čas na jejich dokončení. Zároveň je zde velká úspora spotřebované energie, kterou grafické karty při svém obrovském výpočetním výkonu potřebují. Zanedbatelná není ani nízká pořizovací cena těchto zařízení. Vzhledem k tomu, že paralelní programování na GPU vyžaduje odlišné programování než CPU, musí být vyvíjeny stále složitější kódy pro optimalizaci. Při přihlédnutí k celkové úspoře času při výpočtech se to ale opravdu vyplatí.[1][6]

3 IMPLEMENTACE ALGORITMŮ

Tato kapitola pojednává způsobu programování výpočtů na grafických kartách za pomoci technologie CUDA. Popisuje základy syntaxe, možnosti použití datových typů, problematiku chybových stavů a ošetření více grafických zařízení instalovaných v jednom počítači.

3.1 Programování CUDA aplikací

Jak již bylo řečeno, je kernel základním prvkem architektury CUDA. Kernel se vždy definuje tak, že použijeme v programu klíčové slovo _ _global_ _, přičemž v závorkách předáváme jeho parametry. Pomocí syntaxe <<<M a N>>> je konfigurována mřížka, na které dojde ke spuštění v kernelu. S tím, že M nám udává počet bloků a N počet vláken v každém bloku. Takovýmto způsobem dojde k vytvoření jednorozměrné mřížky, což je při volání kernelu nejsnadnější varianta. [18] Ukázka volání kernelu v jazyce C++ je na obr. 3.1.

```
//Ukázka stanadrdního volání kernelu v jazyce C++
__global__ void mujUkazkovyKernel (int i ) {
// i typu int je zde pamaetrem daného kernelu
}
int main()
{
    // ve fukci main zavoláme náš kernel
    mujUkazkovyKernel<<<M,N>>>>;
    //   pokud bychom za M dosadili například 5 a
    //   za N dosadili 10, vznikla by nám jednorozměrná
    //   mřížka s pěti bloky a každý blok by obsahoval
    //   10 vláken
}
```

Obr. 3.1 Ukázka volání kernelu

Pokud používáme třírozměrné bloky a mřížky musíme použít pro specifikaci proměnné typu dim3. Tento datový typ obsahuje tři složky, které se označují jako x, y, z. Tyto složky konkretizují u mřížek a bloků jednotlivé dimenze.

Pokud deklarujeme v jazyce C kernel, dochází k jistému omezení v porovnání s běžnými funkcemi. Takovýmto omezením je například to, že jako návratový typ musí být vždy použit *void* a také není možné použít proměnný počet argumentů. Při kopírování argumentu kernelu skrze sdílenou paměť z CPU do GPU platí omezení, které stanovuje velikost 256 B pro každý jeden parametr. Rovněž až na výjimky

(GPU s použitím compute capability 3.5) není možno volat z jednoho kernelu další kernely a u deklarace proměnných používat modifikátor static.

Používáme celkem tři typy modifikátorů. Prvním modifikátorem je _ _global_ _, který je volán hostitelem a vykonávám na zařízení. Tento modifikátor se používá pouze pro kernely. Druhým modifikátorem je _ _host_ _, který je opět volán hostitelem ale i prováděn na straně hostitele. Třetím a tedy posledním modifikátorem funkce je _ _device_ _, jenž je volán zařízením a taktéž prováděn zařízením. Tyto tři modifikátory nám tedy definují, kde dochází k uložení kódu funkce, kde bude prováděn a odkud bude volán. Omezení, které platí u deklarace kernelu, tedy vztahující se na návratovou hodnotu void, u dalších funkcí již neplatí a je tedy možné definovat uživatelem i jiné funkce a ty volat přímo z kernelu. Výše uvedené modifikátory _ _device_ _ a _ _host_ _ můžeme spolu kombinovat a díky tomu nám je kompilátor následně schopen vytvořit kód jak pro zařízení, tak pro hostitele. [2]

Datové typy, které se standardně používají, mají svoji obdobu v datových typech jazyka C. Rozdíl je však v tom, že vyjma datového typu ukazatele mají pevně definovanou velikost. Ukazatel má definované 4 bajty pro 32 bitový prostor a 8 bajtů pro 64 bitový prostor. Musíme zde ale rozlišovat zda se jedná o ukazatele do globální paměti zařízení nebo do hlavní paměti hostitele, protože se jedná o naprosto rozdílné prostory pro adresování. Častým řešením je tedy odlišení těchto ukazatelů použitím předpony dev pro GPU paměť a host pro CPU paměť.[16]

Prvním základním datovým typem používaným v prostředí CUDA je *char*, který se používá pro znaky nebo celá čísla a má velikost 1 bajt. Dále je zde *short int* používaný pro celá čísla, která mohou mít maximální velikost 2 bajty. Dalším datovým typem je *int*, který má dvojnásobnou velikost oproti *short int*, tedy 4 bajty. V případě potřeby je možno použít i datový typ *long long int*, který má velikost 8 bajtů. *Float* je dalším datovým typem, který slouží pro práci s čísly v plovoucí řádové čárce a jeho velikost je 4 bajty. Dvojnásobně velký *float* je definován jako double a má tedy velikost 8 bajtů. Jak už bylo zmíněno v předchozím textu, ukazatel *void* může mít velikost 4 nebo 8 bajtů. [4]

Vestavěné proměnné, kterým se také říká automaticky definované proměnné, udávají informace o mřížce kernelu na které byl spuštěn a zároveň jsou zde informace o identifikátorech vláken a to v rámci bloků samotných a nebo bloků, které jsou obsaženy v mřížce. K těmto informacím lze přistupovat v souvislosti s každým jednotlivým vláknem. Funkce, které označují modifikátory _ _global_ _ a _ _device_ _ mají přístup k těmto automaticky definovaným proměnným, přičemž datový typ těchto proměnných je dim3. GridDim je první proměnná udávající velikost mřížky, udává tedy kolik bloků je v jednotlivých dimenzích. Hodnoty obsažené v této

proměnné jsou stejné jako ty, jenž byly definovány programátorem při spuštění kernelu. Proměnná *blockDim* udává velikost bloku v každé jedné dimenzi, přičemž blok je vlastně třírozměrné pole vláken. [3]

Za předpokladu, že chceme v rámci CUDA runtime API provést ošetření chyb, můžeme použít funkci *cudaGetErrorString()*. Tím může být vzniklá chyba lépe popsaná a to pomocí textového řetězce. Díky tomu dochází k lepšímu upřesnění vzniklé chyby než je tomu standartně tj. pomocí kódu *cudaSuccess*, který nám udává, že nedošlo k žádné chybě nebo *cudaError*, který udává, že k nějaké chybě došlo.[2]

Další možností, jak zjistit, která chyba byla jako poslední způsobená při volání nějaké funkce je použití *cudaGetLastError()*. Toto ošetření chyb je vhodné používat téměř vždy při volání funkcí CUDA runtime API. [3]

Pokud je v počítači instalováno více grafických karet, použijeme funkce pro zjištění dostupných zařízení. Funkce *cudaGetDeviceCount (int *devCount)* slouží pro zjištění počtu dostupných zařízení, které podporují aplikace CUDA. Parametr *devCount* ukazuje na hodnotu této funkce uloženou do proměnné. V případě, že v počítači není nainstalováno žádné zařízení, které podporuje CUDA aplikace, dostáváme návratovou hodnotu nula a spolu s ní i chybový kód *cudaErrorNoDevice*. Pokud je verze CUDA driver starší než verze CUDA runtime API, je navrácena chybová funkce *cudaErrorInsufficientDriver*. *CudaGetDeviceProperties* obsahuje informace týkající se zařízení jako je například jméno konkrétního zařízení, velikost globální paměti zařízení, jeho počet multiprocesorů atd. Tyto informace jsou vždy uváděny s daným číslem device pro jednotlivé konkrétní zařízení. Za předpokladu, že konkrétní počítač obsahuje několik zařízení, které mají rozdílné parametry, může si poté daná aplikace vybrat, které z těchto zařízení pro svoje potřeby použije. [17]

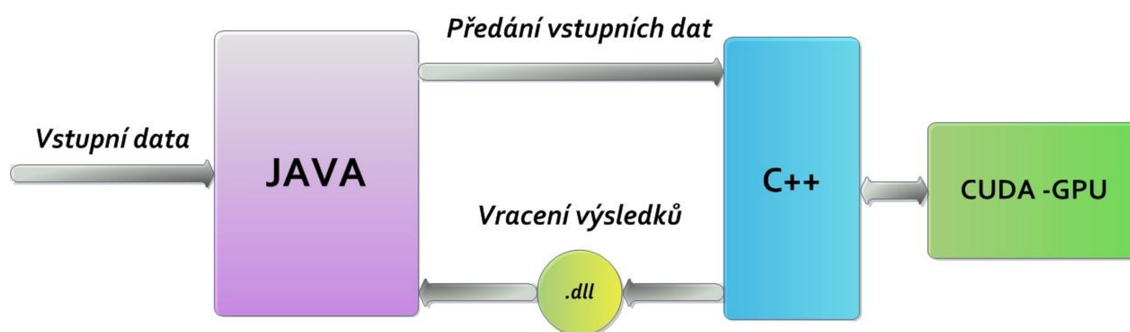
4 SROVNÁNÍ VÝPOČTŮ GPU A CPU

4.1 Program pro demonstraci výpočtů

Pro demonstraci výpočtů na grafické kartě v porovnáním s klasickým CPU počítače byla použita neuronová síť, která se řadí mezi algoritmy umělé inteligence. Jedná se o systém inspirovaný neurony a jejich chováním se v lidském mozku. Tento algoritmus byl vybrán jako nejvhodnější pro zkoumání dat, které poskytnul vedoucí práce.

Tyto data obsahují přibližně šest milionů hodnot, které reprezentují obraz mozku a udávají, zda se jedná o mozek nebo jeho okolí. Nabývají tedy hodnot 1 a 0, kde 1 značí, že se jedná o mozek a 0 že se o mozek nejedná. Data jsou v souboru typu *.csv a na každém řádku je přibližně 370 hodnot které všechny jsou nebo naopak nejsou mozkem. Celkový počet řádků je necelých šestnáct tisíc.

Program jako celek je složen ze tří částí. Jako první je to část programu vytvořená v programovacím jazyce Java, který je objektově orientovaný. Pro programování byl použit software Eclipse. Druhá část programu je vytvořena v jazyce C++, jako programovací prostředí bylo použito Microsoft Visual Studio 2010. Třetí část programu používá architekturu CUDA, která pracuje se stejným programovacím jazykem i prostředím jako druhá část programu.



Obr. 4.1 Propojení jazyků Java a C++

V první části programu(Java) dochází k načítání datového souboru typu csv. V této fázi je do programu načten celý datový soubor(viz. Obr. 4.1), přičemž v programu můžeme měnit počet řádků z načteného souboru, které mají být použity pro trénování neuronové sítě. Je zde také ošetřeno, aby se při trénování používalo stejné množství dat, která reprezentují mozek a těch, která ho nereprezentují. Tím se zamezí tomu, že se pro učení použijí téměř výhradně data, která například mozek nereprezentují a následně nebude možné ostatní data správně klasifikovat vlivem špatného učení.

```java
public static void main(String[] args) throws IOException {

    List<InputData> input = readFromFile("input.csv");
    System.out.println("csv was loaded");
}

public static List<InputData> readFromFile(String fileName) throws IOException {

    BufferedReader fp = new BufferedReader(new FileReader(new File(fileName)));
    int m = 0;

    List<InputData> res = new ArrayList<>();
    while (true) {
        String line = fp.readLine();
        if (line == null) break;
        line = line.replace(",", "");
        StringTokenizer st = new StringTokenizer(line, " \t\n\r\f:");

        InputData item = new InputData();
        m = st.countTokens();
        for (int j = 0; j < m - 1; j++) {
            item.data.add(Double.valueOf(st.nextToken()).doubleValue());
        }
        item.label = Integer.valueOf(st.nextToken()).intValue();

        res.add(item);
    }
    fp.close();

    return res;
}
```

Obr. 4.2 Ukázka vytvořeného kódu pro načtení datového souboru

Vzhledem k tomu, že samotné trénování dat probíhá v další části programu napsané v programovacím jazyce C++, je potřebné, aby první část programu napsaná v jazyce Java předala načtená data do druhé části programu, tedy do C++. To je vyřešeno algoritmem, jehož ukázka je na obrázku 4.3. V zásadě je potřeba rozložit objekty z prostředí Java na jednotlivé proměnné v prostředí C++. Pro tento krok je konkrétně ve Visual Studiu potřeba přidat Java knihovny. To se provede tak, že na náš aktuální projekt klikneme pravým tlačítkem myši a zvolíme možnost Properties. Následně se nám otevře okno s možnostmi nastavení tohoto konkrétního projektu, které se navíc liší podle toho, zda se pro překlad kódu použije možnost Release nebo Debug. Následně se v levém bočním menu v záložce "Configuration Properties" zvolí možnost "VC++ Directories". Zde se do položky "Include Directories" přidají Java knihovny, které jsou obsaženy ve složce, kde je Java nainstalovaná. To je nejčastěji na disku C: ve složce Program Files, pokud je tedy nainstalován operační systém Microsoft Windows.

```
JNIEXPORT void JNICALL Java_cz_fifa_neuron_BrainNetwork_trainNetworkNative
  (JNIEnv *env , jobject object, jobject class0, jobject class1){
    jclass class0class = env->GetObjectClass(class0);
    jclass doubleClass = env->FindClass("java/lang/Double");
    jmethodID sizeMethod = env->GetMethodID(class0class,"size","()I");
    jmethodID getMethod = env->GetMethodID(class0class,"get","(I)Ljava/lang/Object;");
    jmethodID getDoubleValMethod = env->GetMethodID(doubleClass,"doubleValue","()D");
    jint class0size = env->CallIntMethod(class0,sizeMethod);
    jint class1size = env->CallIntMethod(class1,sizeMethod);


    for(int i = 0;i<class0size;i++){
        jobject list = env->CallObjectMethod(class0,getMethod,i);
        jint setSize = env->CallIntMethod(list,sizeMethod);
        vector<double> vec;
        for(int j = 0;j<setSize;j++){
            jobject double_object = env->CallObjectMethod(list,getMethod,j);
            jclass cls = env->GetObjectClass(double_object);

            jdouble value = env->CallDoubleMethod(double_object,getDoubleValMethod);
            vec.push_back(value);
        }
        trida0.push_back(vec);
    }
    for(int i = 0;i<class1size;i++){
        jobject list = env->CallObjectMethod(class1,getMethod,i);
        jint setSize = env->CallIntMethod(list,sizeMethod);
        vector<double> vec;
        for(int j = 0;j<setSize;j++){
            jobject double_object = env->CallObjectMethod(list,getMethod,j);
            jdouble value = env->CallDoubleMethod(double_object,getDoubleValMethod);
            vec.push_back(value);
        }
        trida1.push_back(vec);
    }


    sit.train_network();

}
```

Obr. 4.3 Ukázka zápisu kódu pro předávání dat C++

Zadáním této diplomové práce mimo jiné bylo i to, aby výpočty na grafické kartě byly prováděny z programovacího prostředí jazyku Java. Hlavní programová části 2 a 3 jsou však vytvořeny v C++, proto bylo nutné vyřešit předání funkcí programu z jazyka C++ do Javy. K tomu posloužila možnost vyexportovat výsledný program z jazyka C++ do dynamicky linkované knihovny dll. Tuto knihovnu si pak část programu(Java) načte a může tak pouštět funkce z této knihovny.

Postup pro propojení programů napsaných v jazycích Java a C++ si každý autor může vytvořit sám, v této práci byl postup pro propojení následující. Nejprve je nutné, aby byl v programovacím prostředí jazyku Java obsažen nástroj javah,

který dokáže generovat hlavičkové soubory. Dále je potřeba vytvořit novou třídu s vhodným názvem. Do této třídy se vloží vhodně pojmenované metody (v tomto případě např. *protected native boolean* isBrainNative). Důležité je zde použití klíčového slova native. Následně se správně nastaví a spustí nástroj javah. V programu Eclipse probíhá nastavení a následné spuštění následovně. V horním panelu nástrojů kliknout na Run -> External Tools -> External Tools Configurations . Zde se vytvoří nový nástroj s názvem Run_javah . Do kolonky Location se přidá cesta k souboru javah.exe a do kolonky Working Directory se přidá cesta k aktuální pracovní složce. Následně se do Arguments vepíše "*-jni -verbose -d "${project_loc}${system_property:file.separator}jni" ${java_type_name}* ".

Takto nastavený externí nástroj se uloží a spustí. Tím dojde k vytvoření hlavičkového souboru dříve vytvořené třídy. Následující obrázek 4.4 ukazuje příklad obsahu vytvořeného hlavičkového souboru za pomoci nástroje javah.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class cz_fifa_neuron_BrainNetwork */

#ifndef _Included_cz_fifa_neuron_BrainNetwork
#define _Included_cz_fifa_neuron_BrainNetwork
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     cz_fifa_neuron_BrainNetwork
 * Method:    trainNetworkNative
 * Signature: (Ljava/util/ArrayList;Ljava/util/ArrayList;)V
 */
JNIEXPORT void JNICALL Java_cz_fifa_neuron_BrainNetwork_trainNetworkNative
  (JNIEnv *, jobject, jobject, jobject);

/*
 * Class:     cz_fifa_neuron_BrainNetwork
 * Method:    isBrainNative
 * Signature: (Ljava/util/ArrayList;)Z
 */
JNIEXPORT jboolean JNICALL Java_cz_fifa_neuron_BrainNetwork_isBrainNative
  (JNIEnv *, jobject, jobject);

/*
 * Class:     cz_fifa_neuron_BrainNetwork
 * Method:    isBrainNativeGPU
 * Signature: (Ljava/util/ArrayList;)Ljava/util/ArrayList;
 */
JNIEXPORT jobject JNICALL Java_cz_fifa_neuron_BrainNetwork_isBrainNativeGPU
  (JNIEnv *, jobject, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Obr. 4.4 Hlavičkový soubor vytvořený pomocí javah

Tento hlavičkový soubor nalezneme v adresáři ***jni*** v aktuální pracovní složce. Následně se ve funkci Main přidá instance námi v předchozím bodě vytvořené třídy a zavoláme vytvořenou metodu. V druhé části programu(C++) se potom nastaví cesty k JDK i k nově vytvořeným hlavičkovým souborům. Ve Visual Studiu pomocí Project -> Properties -> C/C++ -> Additional Include Directories. Poté se přidá vytvořený hlavičkový soubor do projektu a to kliknutím pravého tlačítka myši do levé boční lišty, v které jsou soubory projektu a vybráním souboru pomocí Add -> Existing Item. Nakonec se do stavajícího nebo do nově vytvořeného souboru typu *.cpp překopíruje kód z hlavičkového souboru vytvořeného za pomoci javah a kód se následně upraví tak, že se přidá název funkce, vymaže se středník, přidají se složené závorky a případně se kód doplní o názvy parametrů funkce. Tím je vše hotové a nyní se už jen nastaví typ výstupního souboru z jazyka C++ na typ *.dll, který si po kompilaci Java načte.

Druhá část programu obsahuje většinu použitých funkcí a skrze ni se jednak neuronová sít nejprve učí a následně klasifikuje vstupní data do třídy 1 nebo 0. Tyto třídy udávají, zda vstupní data, která se posílají na vstup programu, představují mozek či nikoli. Díky propojení první částí programu, tedy s jazykem Java, není již nutné řešit samotné otvírání a nahrávání vstupního datového souboru. Je zde ale i přesto funkce pro načtení dat na vstup, aby v případě běhu samotného programu bez spouštění přes první část bylo možné tento program korektně spustit. Zároveň je zde také funkce pro načítání trénovacích dat. Tyto dvě funkce se ale při spuštění skrze první část programu nespouští a jejich nastavení tedy nikterak neovlivňuje chování celku.

Nejprve tedy dochází k trénování neuronů za pomocí vstupních dat. Jako trénovací data se používá předem definovaný počet řádku z celého datového souboru. Po spuštění dochází k vytváření náhodného počtu neuronů. Tento počet závisí na počtu řádků, které jsou použity pro trénování. Platí zde téměř přímá úměra, tedy čím více řádků pro trénování použijeme, tím více se vytvoří neuronů. Neurony se přidávají dokud nejsou správně klasifikovány všechny vstupní data z trénovací množiny. Jejich rozdílný finální počet je způsoben rozdílnými vahami, které jsou pokaždé jiné a tak následně buď lépe nebo hůře klasifikují trénovací data.

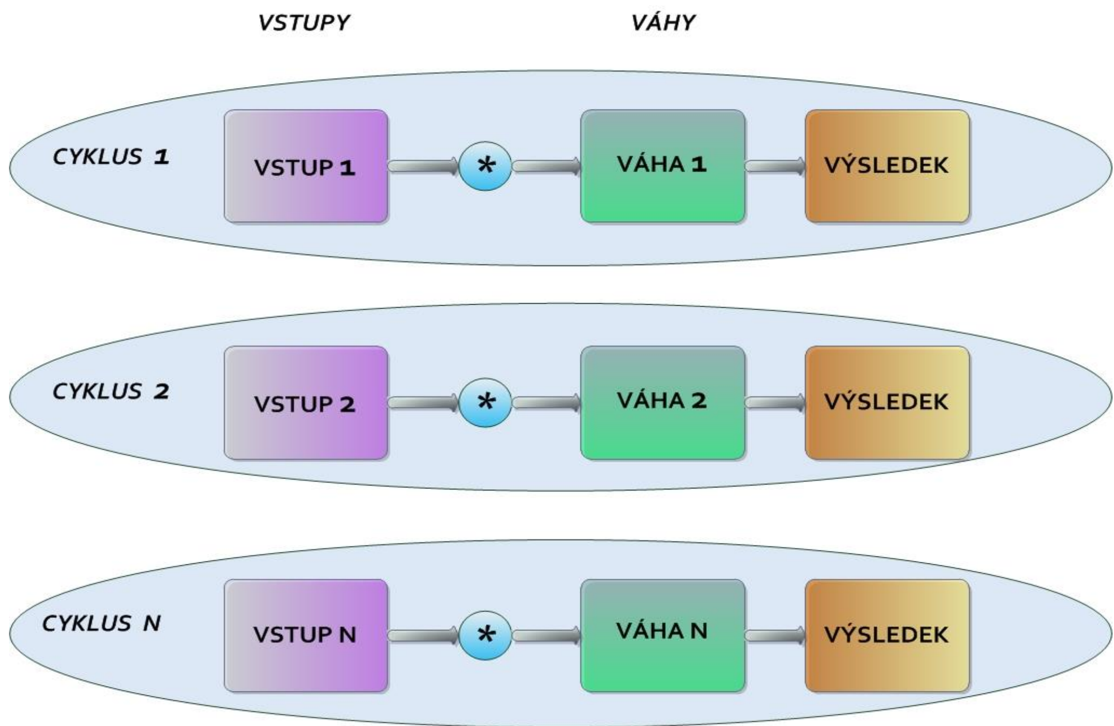
Váhy vytvářených neuronů se přidělují náhodně pomocí funkce rand, jejíž rozsah je vhodně nastavena tak, aby váhy byly co nejvhodnější. V další fázi program testuje, zda jsou neurony správně naučené. To je vyřešeno tak, že neuron musí být schopen správně rozpoznat všechny vstupy z jedné třídy a alespoň jeden z třídy druhé. To znamená, že pokud jeden řádek obsahuje přibližně 370 hodnot, tak správně klasifikuje všech 370 hodnot například ze třídy 0 a minimálně 1 hodnotu z dalšího řádku představující třídu 1. Pokud to nedokáže, tak se váhy přidělí nové a proces se opakuje. Zároveň se odečítají řádky, které vždy patří do třídy 0 nebo 1, použité

pro trénovaní. To se děje dokud se nevyčerpají řádky jedné ze tříd. Takto se postupně vytváří výsledná neuronová síť, kde všechny neurony mají správné hodnoty vah.

Po vytvoření sítě je možné do ní posílat vstupní data a nechat ji aby tyto vstupy klasifikovala. To probíhá tak, že se postupně načítají jednotlivé řádky. Hodnoty na řádku se vynásobí s vahami jednoho neuronu a výsledek každého násobení *hodnota[i] x váha[i]* se uloží a přičte k celkové sumě, v které jsou na konci výpočtu výsledky ze všech násobení vstupních hodnot a vah pro jeden řádek. Tato suma se následně předá další funkci, která do vytvořené neuronové sítě posílá vždy jeden řádek hodnot. Tyto hodnoty projdou přes celou sít, tedy přes všechny neurony a jejich váhy jako v předchozím případě a jejich výsledky se znovu přičítají do proměnné *reakce*. Na konci funkce rozhodne, zda konkrétní řádek hodnot patří do třídy 1 nebo 0.

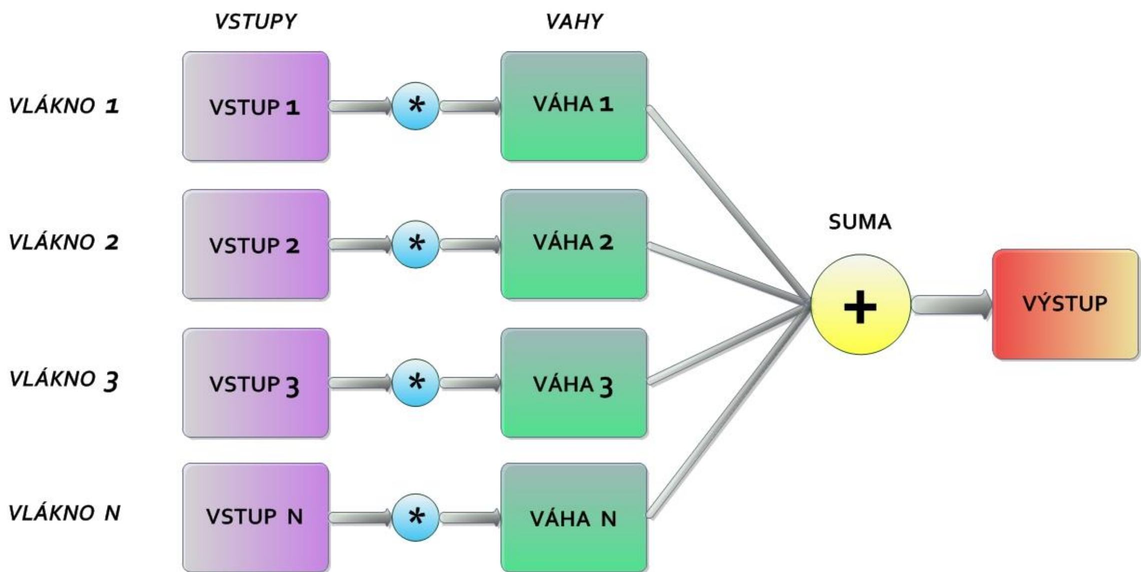
Celý tento proces se provede pro všechny řádky datového souboru, tedy přibližně 16.000x . Výsledek této klasifikace je následně předán jiné funkci, která porovnává, zda tato klasifikace proběhla správně či nikoli. To je možné díky tomu, že vstupní data pro každý jeden řádek obsahují vždy na konci jako poslední hodnotu 1 nebo 0, tedy jestli hodnoty v daném řádku představují nebo nepředstavují mozek. Za pomoci tohoto údaje je tedy možné zaručeně ověřit, zda se klasifikace pro daný řádek zdařila či nikoli.

Třetí částí programu je provádění paralelních výpočtů na grafické kartě za pomocí architektury CUDA, kterou lze implementovat v jazyce C++. V programu jsou dvě části výpočtů, na které lze vhodně aplikovat paralelizaci a tedy za pomoci GPU urychlit výpočet a tak i běh celého programu. Prvním případem je část, kde dochází k výpočtu sum z operace násobení jednotlivých vstupních hodnot s vahami neuronu. Na obrázku 4.5 je ukázka běhu toho výpočtu na klasickém CPU. Z obrázku je zřejmé, že pro výpočet je zapotřebí použití tolika cyklů, kolik je vstupních hodnot v jednom řádku. Program musí nechávat vypočítávat každou hodnotu sumy pro dvojici hodnota to postupně.



Obr. 4.5 Cykly výpočtu na CPU

Na obrázku 4.6 je znázorněn výpočet totožných hodnot ale za pomocí paralelizace na grafické kartě. Zde jednotlivé matematické operace neprobíhají postupně, ale všechny v jednu chvíli za použití vláken, kdy každé vlákno násobí mezi sebou dvojici hodnot a vypočítává výsledek. Výsledky z jednotlivých vláken je následně potřeba spolu sečíst, aby vznikla výsledná suma.



Obr. 4.6 Paralelizace 1 na GPU

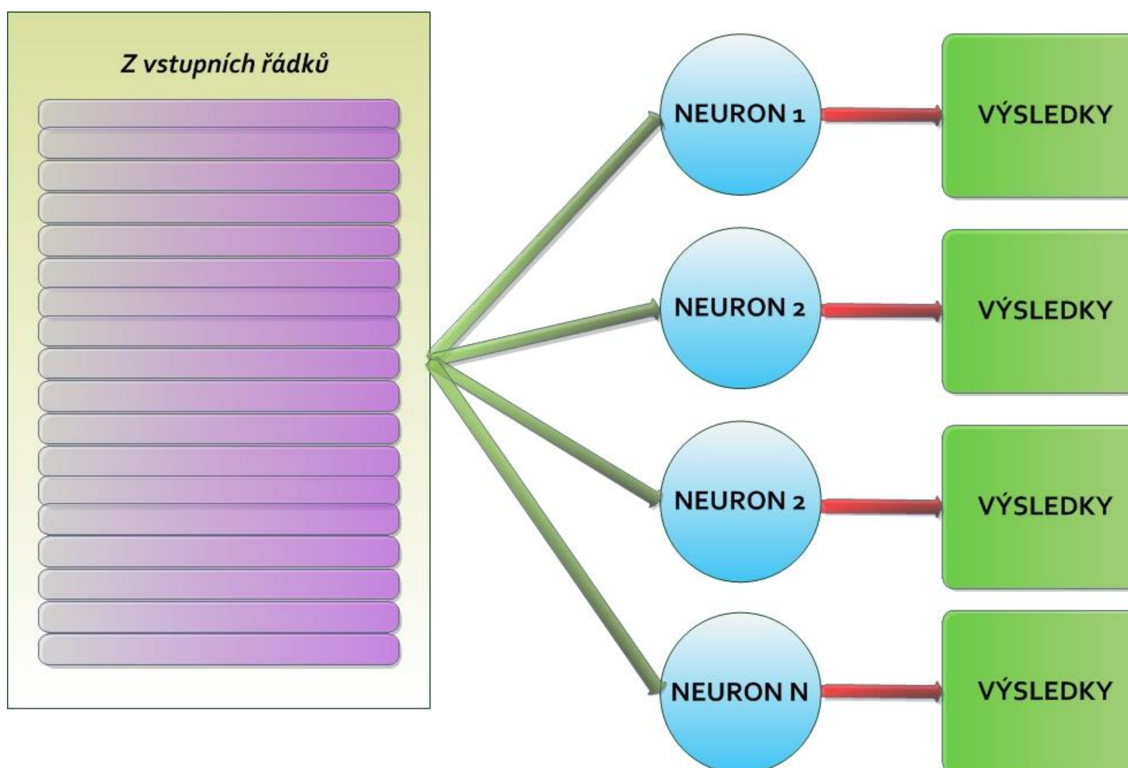
Pro tuto paralelizaci je potřeba si překopírovat vektor s hodnotami vstupů a vah neuronů do globální paměti grafické karty, aby k nim mohly vlákna přistupovat. Ukázka zápisu této paralelizace je na obrázku 4.6.

```
void network::initGPUData(int vstupSize){        // Inicializace paměti na grafické kartě
    cudaMalloc(&d_vstupy, MAX_GPU_INPUTS * vstupSize * sizeof(double));
    cudaMalloc(&d_vahy,  skryta.size() * vstupSize * sizeof(double));
    cudaMalloc(&d_vystupy, MAX_GPU_INPUTS * sizeof(double));
    vector<double> vahy;
    for(int i = 0;i<skryta.size();i++)
        vahy.insert(vahy.end(),skryta[i].vaha.begin(),skryta[i].vaha.end());

    cudaMemcpy(d_vahy, vahy.data(), vahy.size() * sizeof(double), cudaMemcpyHostToDevice);
}

#define BLOCKS 1000
#define N 372
__global__ void Paralelizace(double* vysledek, double* vstup, double* vaha, int vstupsize){
    int inputSetId = blockIdx.y * blockDim.y + threadIdx.y;
    int neuronId = blockIdx.x * blockDim.x + threadIdx.x;

    __shared__ double temp[N];   // Uložení do sdílené paměti pro rychlejší komunikaci paměti s vlákny
    for(int i= 0;i<vstupsize;i++)
        temp[neuronId] += vstup[inputSetId*vstupsize+i] * vaha[neuronId*vstupsize+i];

    __syncthreads();   //Pararedukce
    for (unsigned int s = N/2; s>0; s >>= 1){
        if (neuronId < s)
            temp[neuronId] += temp[neuronId + s];
    }
    __syncthreads();
    if(inputSetId == 0)
        vysledek[inputSetId] = temp[0];


}
```

Obr. 4.7 Ukázka kódu paralelizace 1

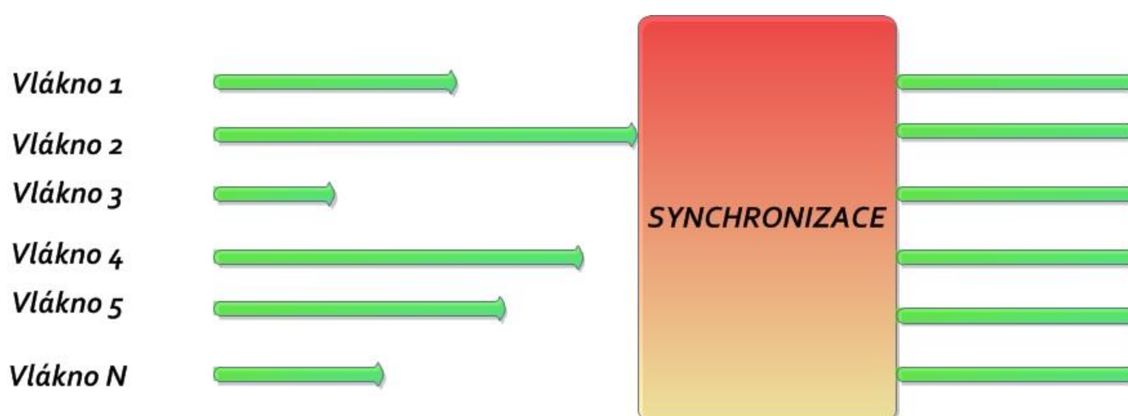
Druhou paralelizací, kterou lze díky použití architektury CUDA, je výpočet reakcí všech neuronů v sestavené neuronové sítí na každý řádek vstupní množiny dat a to současně. To znamená, že místo postupného počítání reakcí každého neuronu v síti na jednotlivé řádky vstupních dat je možné nechat všechny neurony počítat tyto reakce a tím urychlit výpočty. Při těchto operacích však grafická karta musí vždy načítat data v rámci své paměti ale data vyměňovat mezi svou pamětí a pamětí počítače.

Nedá se tedy jednoduše říct, že při síti obsahující 200 neuronů které zpracovávají 16.000 řádku je úspora rovna 200 x 16000 cyklů, protože to není z výše uvedených důvodů pravda. Obrázek 4.7 znázorňuje vhodné použití paralelizace v tomto případě.



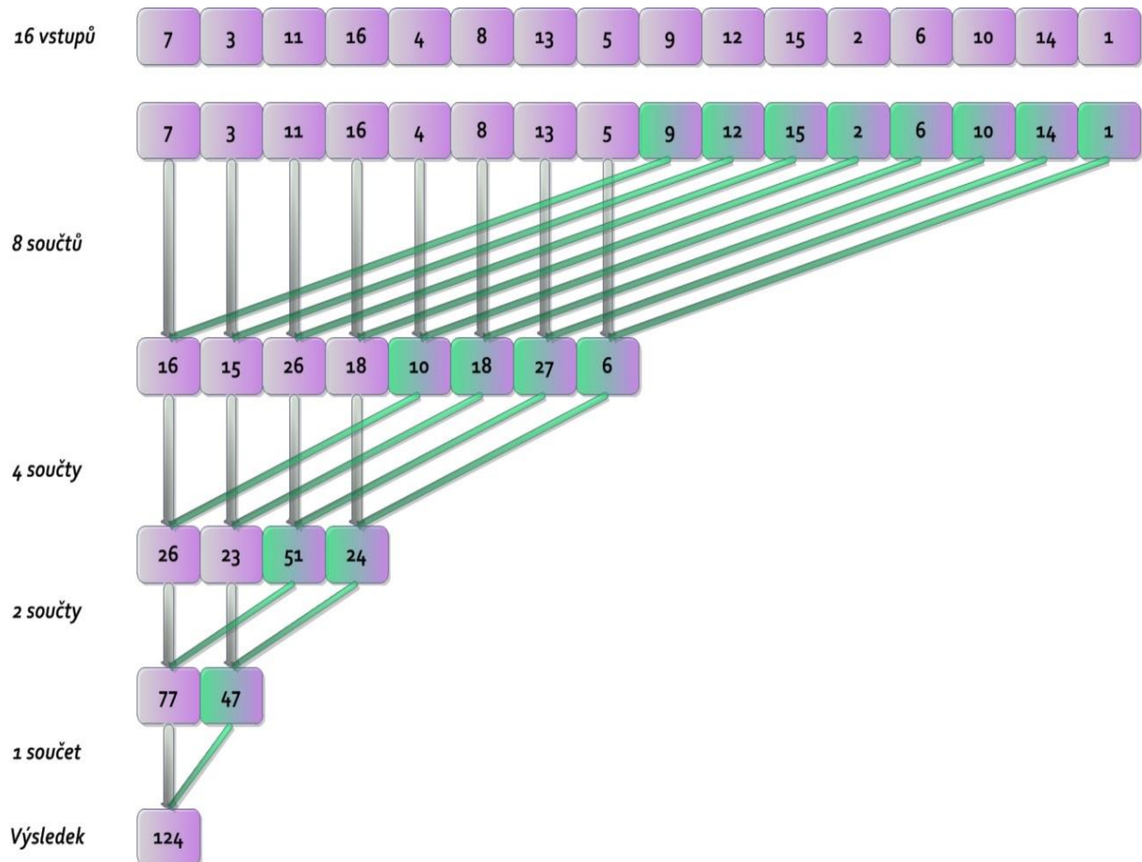
Obr. 4.8 Znázornění paralelizace 2

Pro dosažení celkového výsledku ze všech vláken, v kterých současně probíhaly výpočty, je nutné zajistit nejprve jejich správné doběhnutí a ukončení. Výpočty prováděné nad daty nejsou totiž pro jednotlivé hodnoty vstupních dat naprosto shodně dlouhé. Pro takovéto případy má CUDA funkci s názvem _syncthreads(), která vždy počká, až doběhnou úlohy ve všech vláknech a až následně dojde k předání výsledků. Tento proces ukazuje obrázek 4.8



Obr. 4.9 Ukázka funkce synchronizace vláken

Na obrázku 4.9 je znázorněna ukázka paralelní redukce. Tu je nutné provést v případě, že se z výsledků jednotlivých vláken musí vytvořit celkový výsledek. Dosáhneme toho pomocí postupného sčítání výsledků z vláken mezi sebou, až zůstane jeden jediný výsledek, který již není s čím sečíst.



Obr. 4.10 Ukázka principu pararedukce

Vytížení počítače bylo sledováno pomocí systémového nástroje Správce úloh, který umožňuje sledování vytížení procesoru i paměti RAM a to s relativně rychlou obnovovací frekvencí sledování změn.

Pro sledování zatížení grafické karty byl použit volně stažitelný program s názvem GPU-Z od společnosti TechPowerUp, který měří mnoho parametrů grafické karty jako je například vytížení paměti, vytížení GPU apod. Tento program svoje hodnoty obnovuje přibližně každou sekundu a tak nebylo vždy možné přesně změřit hodnoty zatížení karty. Jiné programy pro měření zátěže grafického i na desce osazeného procesoru nebo pro využití paměti RAM počítače a grafické karty nebyly při tvorbě této diplomové práce použity.

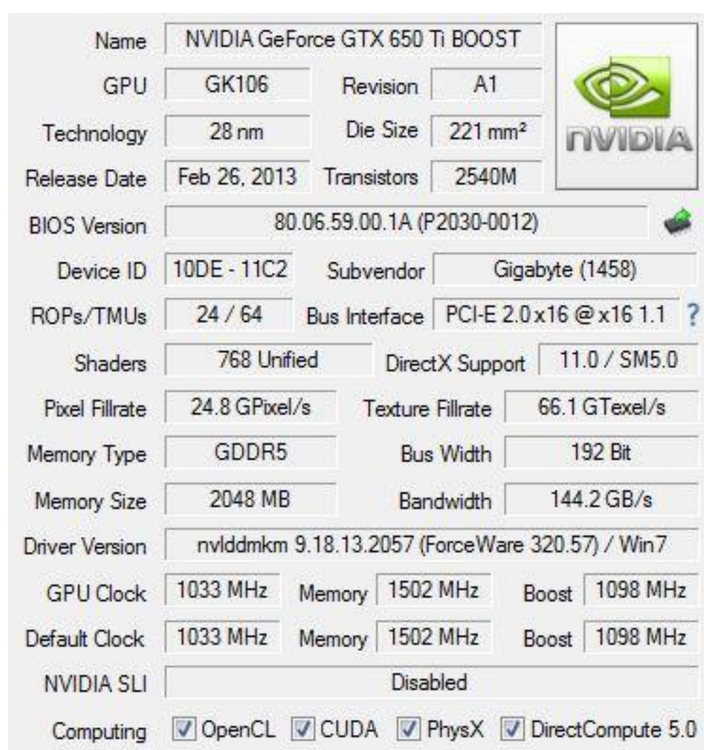
Na obrázku 4.2 je ukázka výpisu z programu, puštěném ve MS Visual Studiu, v které byl program vždy spouštěn pro testování. Program nejprve vypisuje zda načítá soubor, v případě že se mu soubor nepodaří otevřít, vypíše chybovou hlášku. Následně začne vypisovat postupné přidávání neuronů. Když jsou všechny neurony natrénovány a je vytvořena celá neuronová sít, tak se ve výpisu programu objeví informace o natrénování sítě a také o tom, kolik bylo při tomto trénovaní celkově vytvořeno neuronů. Následně dojde ke klasifikaci zpracovávané GPU a hned poté CPU. Ve výpisu programu tak vidíme časy, jaké potřeboval procesor počítače a grafická karta na klasifikaci vstupních dat. To kolik řádků dat bylo pro tento běh programu nastaveno zde najdeme též. Je zde také vypsáno, kolik bylo správně klasifikovaných vstupních řádku z použitého datového souboru. Pro zpřehlednění a usnadnění zjištění výsledků použité paralelizace výpočtů na grafické kartě je zde údaj o násobcích urychlení běhu na GPU o proti době běhu na CPU, který se automaticky vypočítává jako podíl času výpočtu na CPU a času výpočtu na GPU.

```
Pridavam neuron. Zbyva 8 + 8
Pridavam neuron. Zbyva 8 + 4
Pridavam neuron. Zbyva 7 + 4
Pridavam neuron. Zbyva 5 + 4
Pridavam neuron. Zbyva 4 + 4
Pridavam neuron. Zbyva 4 + 3
Pridavam neuron. Zbyva 3 + 3
Pridavam neuron. Zbyva 3 + 2
Pridavam neuron. Zbyva 1 + 2
Pridavam neuron. Zbyva 1 + 0
Sit natrenovana. Bylo vytvoreno 224 neuronu
Zpracovano 5000/15945
GPU time: 94 ms
Spravne klasifikovano 3549 z 5000 vstupu

CPU time: 1201 ms
Zrychleni: 12x
Pokračujte stisknutím libovolné klávesy...
```

Obr. 4.11 Výpis z programu

4.2 Dosažené výsledky

Při tvorbě této práce byl používán počítač se čtyřjádrovým procesorem Intel Core 2 Quad Q8400@2,66GHz, pamětí RAM o použitelné velikosti 8GB a s operačním systémem Windows 7. Grafická karta osazená v počítači byla nVidia GeForce GTX 650 Ti Boost, která podporuje technologii CUDA. Parametry grafické karty popisuje následující obrázek 4.3.



Obr. 4.12 Parametry použité grafické karty

Pro demonstrační zrychlení výpočtů s použitím procesorů na grafické kartě bylo použito celkem 16 nastavení vstupních řádků z datového souboru. Tyto nastavení řádků, které se v běhu programu použijí jako data pro klasifikaci, jsou v rozsahu 1000 až 15945. Pro trénování bylo u všech počtů vstupů nastaveno 500 řádků jako trénovacích. Počty vytvořených neuronů byly vždy různé. Důvody pro tento jev jsou popsány výše v této práci. Jejich počet se v průměru pohyboval okolo 230 vytvořených. Nastavení hodnot pro demonstraci bylo použito takové, aby bylo co nejlépe poznat chování grafické karty při jejím použití v pozici výpočetní jednotky. Takovýmto chováním je fakt, že výpočty na grafické kartě jsou při použití malého množství vstupních dat pro výpočty jen o malý násobek rychlejší

než při použití CPU. Může se i stát, že čas výpočtu na GPU je nižší než čas na CPU. To je způsobeno tím, že grafická karta si musí z paměti počítače nahrát data ke zpracování do své vlastní paměti, následně nad nimi provede rychlý výpočet, ale musí je předat zase zpět paměti počítače. Tímto kopírováním dat z jedné paměti do druhé dochází k jisté časové prodlevě ta má za následek to, že procesor počítače sice samotné výpočty provede o poznání pomaleji než procesory na grafické kartě, ale díky okamžitému přístupu do paměti s daty je celkový čas strávený nad výpočty nižší, než jaký je na kartě. Vysoký výkon GPU se začne projevovat až v momentě, kdy se provádí stále větší množství výpočtů nad stále objemnějším množstvím dat. Tyto data ale jsou již nahrána v paměti grafické karty a tak nemusí docházet k neustálému kopírování mezi pamětmi. Hodnoty, které byly použity, jsou vypsány v následující tabulce.

Tab. 4.1 Porovnání výsledků CPU a GPU

Počet řádků pro klasifikaci	Počet řádků pro trénovaní	Počet vytvořených neuronů	Čas výpočtu na CPU [ms]	Čas výpočtu na GPU [ms]	Urychlení v násobcích
1000	500	230	249	45	5,53
2000	500	220	370	47	7,87
3000	500	251	537	59	9,10
4000	500	210	819	75	10,92
5000	500	237	1097	94	11,67
6000	500	227	1350	109	12,39
7000	500	231	1694	125	13,55
8000	500	213	1929	140	13,78
9000	500	243	2231	161	13,86
10000	500	224	2371	169	14,03
11000	500	219	2668	187	14,27
12000	500	231	2913	203	14,35
13000	500	260	3310	212	15,61
14000	500	245	3604	227	15,88
15000	500	259	4273	265	16,12
15945	500	257	4805	281	17,10

V prvním sloupci tabulky jsou uvedeny použité množství řádků ze souboru vstupních dat pro výpočet, v druhém sloupci počet nastavených trénovacích řádků a ve třetím počet vytvořených neuronů v rámci trénovaní a výstavby neuronové sítě. Třetí a čtvrtý sloupec obsahuje časy výpočtů klasifikace na CPU a GPU. Ve sloupci

s názvem Urychlení v násobcích nám hodnoty udávají, kolikrát se zvýšila rychlost výpočtu na GPU v porovnání s CPU.

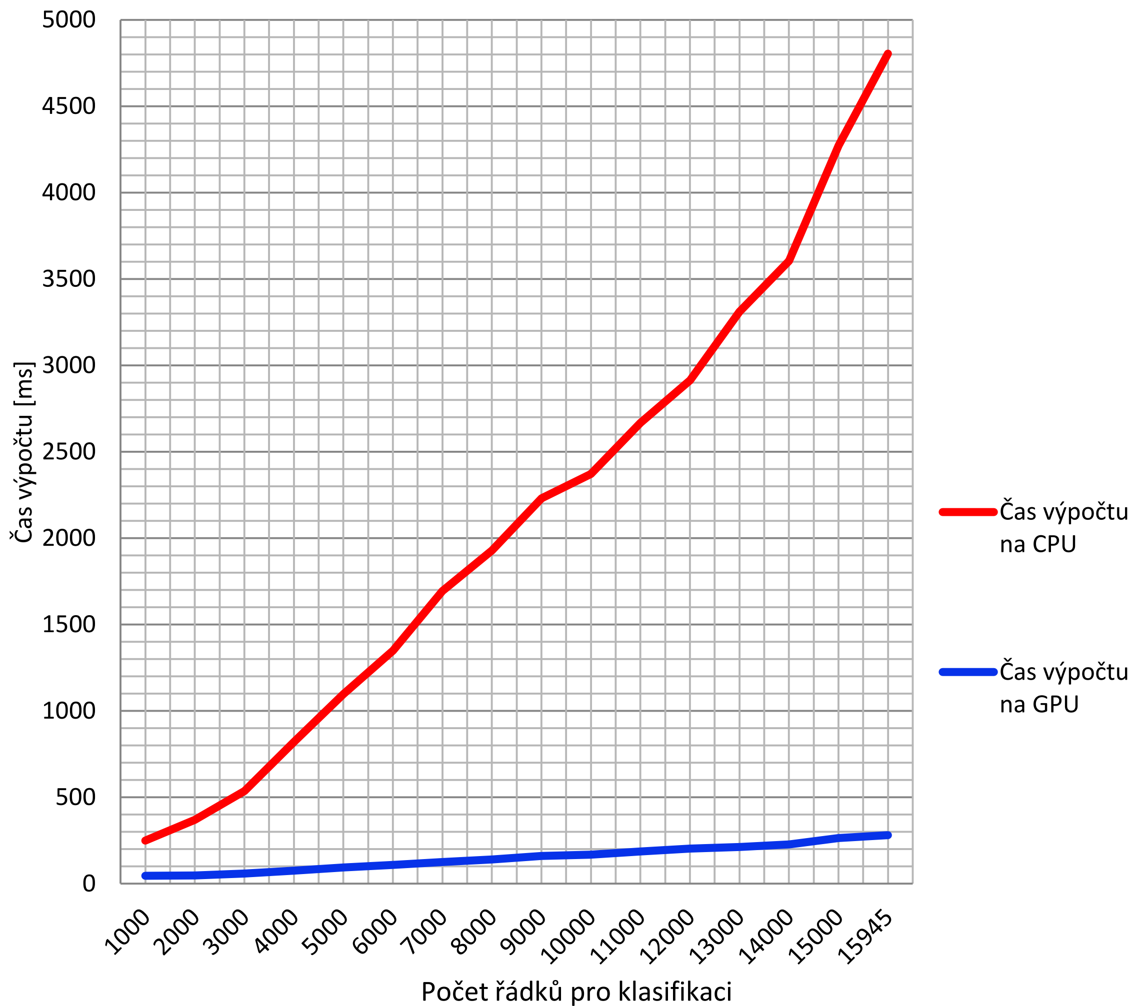
Následující graf ukazuje dobu trvání výpočtů na GPU a CPU při použití konkrétních hodnot pro výpočet. Při nižších hodnotách nastavení klasifikačních řádků je čas, který potřebuje GPU pro výpočty 5,5x menší něž u CPU, ale se zvyšujícími se použitými hodnotami pro výpočet rostě čas na CPU rychle téměř lineárně nahoru a čas potřebný pro GPU rostě jen velice mírně.

Největší rozdíl času pro výpočet na GPU a CPU byl 281ms a 4805ms, což představuje sedmnácti násobek potřebného času. K největšímu přiblížení časů dochází při použití nejnižšího počtu klasifikačních řádků, kdy GPU má čas výpočtu 45ms a CPU 249ms.

Grafická karta potřebuje pro nejvyšší měřený výpočet přibližně 80MB své operační paměti a počítač jí využije po odečtení od objemu dat pro celkový běh programu zhruba 90MB. Toto měření ale může být trochu zavádějící. Musíme totiž přihlédnout k faktu, že počítač používá svou paměť pro všechny procesy a jejich data, které obsluhuje. Grafická karta se samozřejmě stará vždy i o zobrazování toho, co na počítači děláme, ale vliv na využitou paměť je za předpokladu vypnutí všech graficky náročných aplikací malý. Využití procesoru bylo vždy na hodnotě 25%, což představuje plné využití jednoho z jeho čtyř jader.

Program není na CPU paralelizován mezi více procesorů, proto se ani jiné hodnoty dosáhnout nedalo. Grafická karta odolávala při výpočtech zátěži přibližně 15%, což nepředstavuje nějak závratné zatížení. Jak již bylo zmíněno dříve, nástroj pro měření grafického výkonu GPU-Z nedokáže obnovovat svoje hodnoty v pro tento případ vhodném časovém intervalu, takže tato hodnota zátěže grafické karty nemusí být zcela přesná.

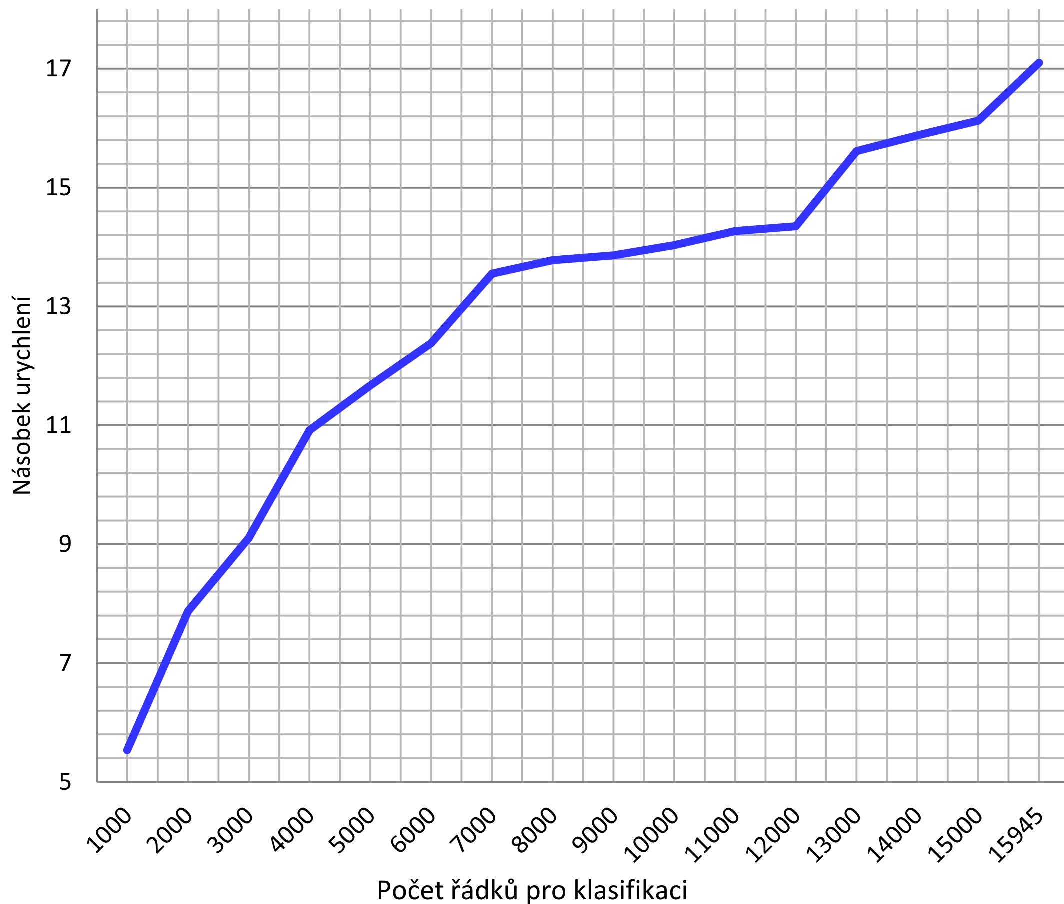
Graf 4.1 Závislost hodnot výpočtu na době trvání výpočtů CPU a GPU



Graf 4.1 znázorňuje, k jak velkému urychlení výpočtu došlo při počítání těch samých hodnot na grafické kartě a na počítači, přičemž hodnota 1 vyjadřuje stav, kdy je potřeba stejný čas pro výpočet jak na GPU tak na CPU. Z grafu je patrné, že při výpočtu nižších hodnot, tedy při menším objemu vstupních dat se časy výpočtu na grafické kartě a procesoru přibližují a rychlost výpočtů na CPU se při zvyšujícím se objemu vstupních dat výrazně snižuje. Tento jev je způsoben tím, že až u většího množství dat se naplno projeví paralelizace výpočtů, která se odehrává na grafické kartě.

Maximální hodnota urychlení, které bylo dosaženo, je rovna sedmnácti násobku času, který potřebuje CPU v porovnání s GPU pro výpočet stejného počtu vstupních řádků z datového souboru.

Graf 4.2 Závislost urychlení výpočtů pomocí GPU na počtu řádků pro klasifikaci



Z grafu 4.2 je patrné, že se hodnota urychlení téměř lineárně zvyšuje s přibývajícími vstupními řádky, které neuronová síť klasifikuje. Měření celého programu může však vždy být ovlivněno mnoha nevyžádanými procesy, které se při běhu počítače spouští a tak může docházet k výkyvům časů pro výpočty. Pro každé nastavení programu bylo provedeno několik měření. Časy uvedené v tabulce jsou průměrnými hodnotami, kterých bylo dosaženo během opakovaného spouštění programu pro konkrétní hodnoty.

Klasifikace, která byla prováděna nad vstupními daty dosahovala hodnot v rozmezí 50-75% a to vzhledem k použitému počtu řádků pro učení, kdy při nižším počtu řádků dosahovala klasifikace nižších hodnot a při použití většího počtu trénovacích dat se jednalo o procenta vyšší.

V následující tabulce 4.2 jsou uvedeny dosažené hodnoty pro klasifikaci vstupních dat. Bylo použito šest hodnot pro nastavení počtu trénovacích řádků, na kterých se algoritmus umělých neuronových sítí učí rozpoznávat vstupní data a řadit je správně do tříd. Pro konkrétní nastavení trénování byla použita vždy čtveřice hodnot představující počet použitých řádků ze vstupního datového souboru, nad kterými byla prováděna samotná klasifikace.

Tab. 4.2 Výsledky správné klasifikace vstupních dat

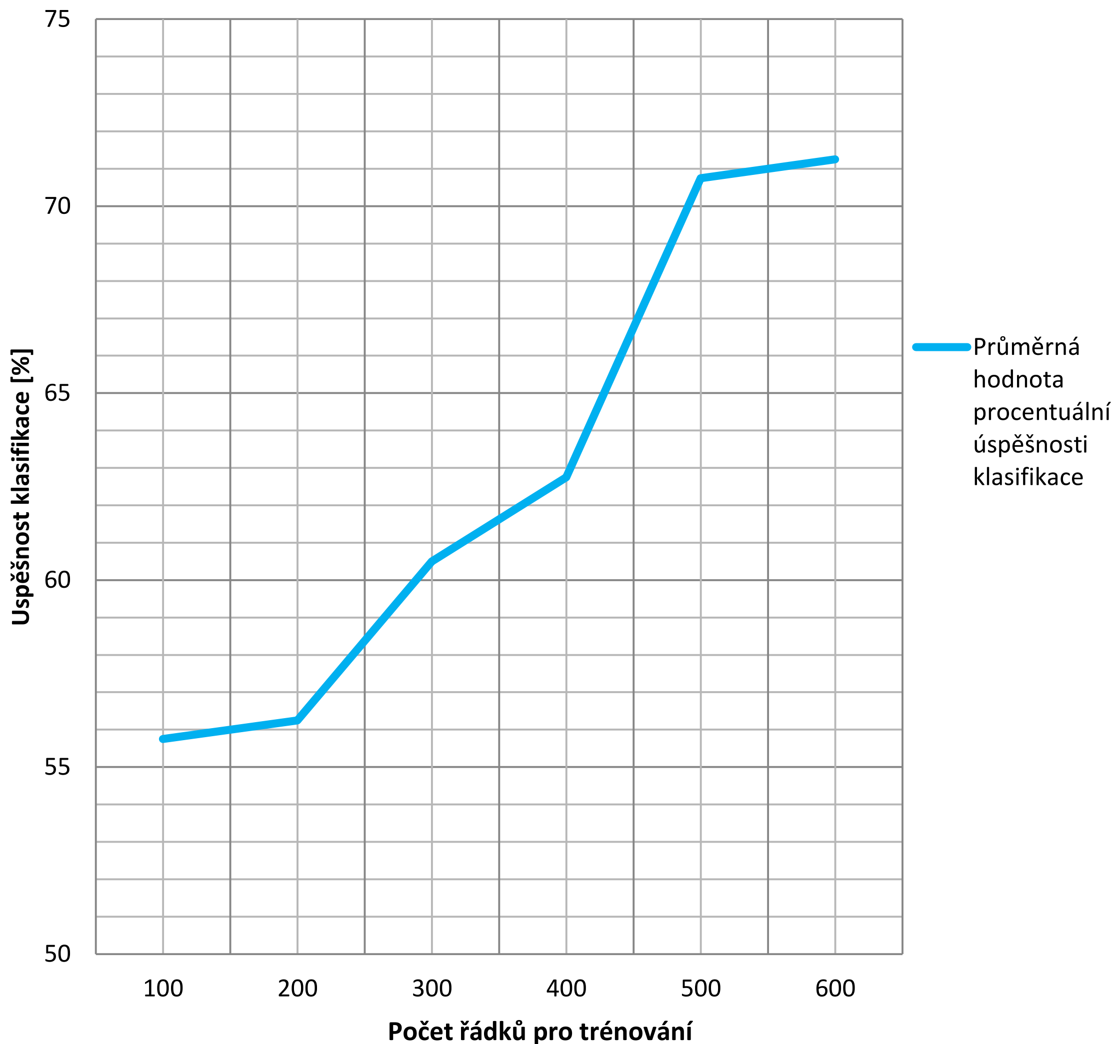
Počet řádků pro klasifikaci	**Počet použitých vstupních řádků pro trénovaní**					
	100	**200**	**300**	**400**	**500**	**600**
	Úspěšnost klasifikace vstupních dat [%]					
1000	47	54	59	68	71	72
5000	55	57	52	60	72	73
10000	62	60	69	53	69	72
15000	59	54	62	70	71	68
Průměrně	***55,75***	***56,25***	***60,5***	***62,75***	***70,75***	***71,25***

Hodnoty procentuální úspěšnosti klasifikace se často náhodně mění, proto bylo vždy provedeno více měření pro jedno a to samé nastavení programu. Poslední řádek tabulky udává, jaká byla průměrná procentuální hodnota správně klasifikovaných dat a to vždy pro všechny čtyři hodnoty počtu řádků určených ke klasifikaci vztažené k jedné nastavené trénovací hodnotě.

Na následujícím grafu 4.3 je zobrazena závislost počtu řádků použitých pro trénování na procentuální úspěšnosti následné klasifikace. Z grafu je patrně, že s použitím většího množství řádků pro učení roste hodnota správně klasifikovaných dat.

Zvyšováním počtu řádků pro učení roste čas potřebný pro natrénování použité neuronové sítě. Proto byla u demonstračních měření doby výpočtů GPU a CPU, jejichž výsledky jsou zaznamenány v tabulce 4.1, použita v této práci hodnota pěti set trénovacích řádků, protože s použitím tohoto množství řádků použitých pro učení dosahuje program nejlepšího poměru správně klasifikovaných vstupních dat k času, který potřebuje pro natrénovaní neuronové sítě. To je také patrné z grafu, kde se při použití pěti set řádků dosahuje jen přibližně o jedno procento horší celkové klasifikace něž pokud je použito o sto řádků více.

Graf 4.3 Závislost úspěšnosti klasifikace na použitém počtu řádků pro trénování



Pro účely této práce a demonstraci použití grafické karty jako superpočítače však tyto dosažené výsledky správné klasifikace nejsou tak podstatné. Tato práce se nezabývala natrénováním přesného modelu pro detekci částí mozku, ale ověřením urychlení použitého algoritmu za pomocí paralelizace na GPU. Důvodem k nižším procentům úspěšné klasifikace může být i zdroj vstupních dat, který za předpokladu špatných hodnot v jednotlivých řádcích může způsobovat následnou špatnou programovou klasifikaci.

Jak již bylo uvedeno výše v textu, byl program při testování spouštěn s programového prostředí Microsoft Visual Studia. Důvod pro tuto skutečnost byl takový, že pokud se program spouští z prostředí Eclipse, tedy z jazyku Java, dochází vlivem předávání dat mezi jazyky a navíc ještě mezi pamětí grafické karty

a pamětí počítače k různým menším či větším zpožděním. To má za následek delší běh výpočtů na GPU díky kterým potom porovnání výpočetních výkonů GPU a CPU není zcela přesné.

Také výpisy programu při použití propojení s Javou se často nezobrazují ve zcela správném pořadí, ale vždy se zobrazí všechny informace, které měly být programem vypsány. To může být opět způsobeno předáváním dat v rámci programu, kdy potom dojde ke zpoždění při výpisu požadovaného informačního textového řetězce. Tento problém se nepodařilo přes veškerou snahu odstranit, ale vzhledem k tomu, že se jedná spíše o drobnou kosmetickou záležitost, nemá na nic žádný vliv.

ZÁVĚR

Tato diplomová práce se zabývala realizací superpočítače pomocí grafické karty. Pro tyto potřeby byl použit jazyk Java a C++ spolu s technologií CUDA, která umožňuje provádět programování algoritmů pro GPU a následně zabezpečuje jejich samotné vykonávání na grafické kartě.

Základním požadavkem na tuto diplomovou práci bylo nastudování problematiky výpočtů na grafických kartách, seznámení se s technologiemi CUDA a OpenCL a také algoritmy umělé inteligence a jejich implementace do prostředí jazyku Java.

Toto vše bylo splněno a následně byl aplikován vytvořený algoritmus umělé inteligence pro výpočty jak na CPU tak na GPU, jehož výsledky byly zaznamenány do tabulky a grafů. Algoritmus umělé inteligence byl v této diplomové práci reprezentován neuronovou sítí, která se nejprve ze vstupních dat natrénuje a následně klasifikuje nově příchozí data dle dříve naučených vah.

Hlavní přínos této práce spočívá ve vytvoření takového algoritmu umělé inteligence, který ukazuje srovnání výkonnosti při výpočtech na grafických kartách oproti standardním výpočtům za pomoci procesoru počítače. Dosažené výsledky při výpočtech potvrzují teoretická tvrzení, že s použitím grafických karet pro výpočty dosáhneme zrychlení celého procesu a dojdeme tak rychleji ke správnému výsledku početních operací. V této diplomové práci bylo dosaženo sedmnácti násobného urychlení výpočtů při použití GPU o proti klasickému CPU. Tato hodnota by pravděpodobně stále lineárně rostla při použití většího datového souboru, než toho který poskytnul pro účely demonstrace vedoucí práce.

Vytvořený algoritmus by mohl sloužit pro klasifikaci většího množství vstupních dat například biomedicínského charakteru. Vytvořené propojení mezi programovacími jazyky Java a C++ lze použít i pro jiné programy a to pouze s nezbytnými úpravami. Na zváženou je, zda by pro propojení architektury CUDA s jazykem Java nebylo vhodnější použít platformu jcuda, která vytváří jejich přímé propojení. Složitost kódu s jejím použitím by byla pravděpodobně menší a nebylo by nutné vytvářet propojovací prvek mezi jazyky C++ a Java, který v případě této práce představoval dynamicky linkovanou knihovnu obsahující funkce z programu psaného v C++.

Používání grafických karet osazených velkým množstvím procesorů na výpočetní operace se zdá být do budoucna vhodné pro mnohé případy zpracovávání objemných dat. Šifrovací algoritmy patří do oblasti, která se rychle rozvíjí a jejich složitost stále roste. Pro šifrování a dešifrování je zapotřebí stále většího výpočetního výkonu a to nemluvě o případném prolamování(pouze z důvodu ověření účinnosti a odolnosti) takových vytvořených šifrovaných posloupností.

Nejmodernější výrobní technologie dovolují stále snižovat nejen velikost vyráběných zařízení, ale mnohdy snižují i jejich cenu. V dnešní době je standardem, že se u výběru téměř čehokoli hledí hlavně na cenu a kvalita často zůstává v žebříčku priorit až kdesi na konci. Grafické karty nejnovějších generací nabízí obrovský výkonnostní potenciál a to za velice nízkou cenu ve srovnání s klasickým výpočetním hardwarem, který nedosahuje výkonnosti GPU jader a přitom se jeho cena pohybuje několikanásobně výš, než za kolik se dnes dá pořídit výkonná grafická karta. Přesunutí výpočetních operací na GPU tedy není jen pouhou zajímavou novinkou, kterých se v posledních letech objevuje v oblasti informačních systémů stále více, ale pro svoji závratnou cenu se nemají ve finále možnost uchytit v praxi. Patří naopak do skupiny těch nových technologií a nápadů, kterých je sice jen pomálu, ale přináší něco nového, dobře a efektivně použitelného a zároveň cenově přístupného pro každého.

Literatura

[1] ASCHER, Uri M.; RUUTH, Steven J.; WETTON, Brian TR. Implicit-explicit methods for time-dependent partial differential equations. *SIAM Journal on Numerical Analysis*, 1995, 32.3: 797-823.

[2] COOK, Shane. *CUDA programming: a developer's guide to parallel computing with GPUs*. Boston: Elsevier, MK, c2013, xiv, 576 p. ISBN 01-241-5933-8.

[3] FARBER, Rob. *CUDA application design and development*. Amsterdam: Elsevier, c2011, xvii, 315 s. ISBN 978-0-12-388426-8.

[4] HWU, Wen-mei W. *GPU Computing Gems Jade Edition (Applications of GPU Computing Series)*. Morgan Kaufmann, 2011 [cit. 2014-05-16]. ISBN 978-0123859631.

[5] JAKLIN, Petr. Neuronové sítě. [online]. [cit. 2013-12-30]. Dostupné z: cgg.mff.cuni.cz/~pepca/prg022/mucha/

[6] JANUSZEWSKI, M.; KOSTUR, M. Accelerating numerical solution of stochastic differential equations with CUDA. *Computer Physics Communications*, 2010, 181.1: 183-188.

[7] KIRK, David a Wen-mei HWU. *Programming massively parallel processors: a hands-on approach*. Burlington: Morgan Kaufmann Publishers, 2010. ISBN 978-0-12-381472-2.

[8] KLASCHKA, Jan a Emil KOTRČ. *Klasifikační a regresní lesy*. ROBUST, 2004.

[9] KOBRTEK, Jozef. *Využití grafického procesoru jako akcelerátoru - technologie OpenCL: Use of GPU as Accelerator - Technology OpenCL*. Brno: Vysoké učení technické, Fakulta informačních technologií, 2010 [cit. 2014-05-07].

[10] KORČUŠKA, Robert. *Využitelnost knihovny CUDA v praktickém zpracování obrazů*. 2013, 55 l.

[11] KOSSIDA, Sophia.International Journal of: Systems Biology and Biomedical Technologies (IJSBBT). 4. vyd. Information Resources Management Association, 2013. ISSN 2160-9586. Dostupné z: http://www.igi-global.com/article/state-of-the-art-gpgpu-applications-in-bioinformatics/105596

[12] LEGEŇ, Michal. *Implementace a testování hashovacího algoritmu MD5: Implementation and testing of MD5 hash algorithm*. Brno: Vysoké učení technické, Fakulta elektrotechniky a komunikačních technologií, 2008

[13] PACURA, Dávid. *Paralelizace výpočtů pomocí GPGPU prostředků* [online]. 2013, 60 l. [cit. 2014-05-18]. Dostupné z: https://dspace.vutbr.cz/bitstream/handle/11012/27793/PARALELIZACE%20V%C3%9DPO%C4%8CT%C5%AE%20POMOC%C3%8D%20GPGPU%20PROST%C5%98EDK%C5%AE.pdf?sequence=1&isAllowed=y

[14] PĚCHOTOVÁ, Barbora. *SROVNÁNÍ KLASIFIKAČNÍCH METOD PRO APLIKACI NA BIOLOGICKÝCH DATECH* [online]. Brno, 2009 [cit. 2013-12-30]. Dostupné z: is.muni.cz/th/243713/prif_b/BP_Pechotova.doc

[15] PERCIVAL, C.; JOSEFSSON, S.: The scrypt Password-Based Key Derivation Function. Draft, Internet Engineering Task Force, 2012. Dostupné z : http://tools.ietf.org/html/draft-josefsson-scrypt-kdf-01

[16] POTĚŠIL, Josef. *Využití moderních GPU pro obecné výpočty: General Purpose GPU*. Brno: Vysoké učení technické, Fakulta informačních technologií, 2009 [cit. 2014-04-24].

[17] SANDERS, Jason. *CUDA by example: an introduction to general-purpose GPU programming*. 1st print. Upper Saddle River: Addison-Wesley, c2011, xix, 290 s. ISBN 978-0-13-138768-3.

[18] ŠIMEČEK, Ivan a Jaroslav SLOUP. *Programování grafických akcelerátorů*. 1. vyd. V Praze: České vysoké učení technické, 2013, 138 s. ISBN 978-80-01-05195-5.

[19] VÁCLAVÍK, Jiří. *Paralelní zpracování signálů pomocí GPU* [online]. 2013, 33 l. [cit. 2014-03-17]. Dostupné z : https://dspace.vutbr.cz/xmlui/bitstream/handle/11012/27373/Zaverecna%20prace%20Valcavik.pdf?sequence=2&isAllowed=y

[20] WILT, Nicholas. *The CUDA handbook: a comprehensive guide to GPU programming*. 1st printing. Upper Saddle River: Addison-Wesley, 2013, xxv, 494 s. [cit. 2014-05-22]. ISBN 978-0-321-80946-9.

[21] ZOU, Hang; WANG, Hua-qiu; HUANG, Yong. GPU Accelerated Rainbow Tables Analysis of MD5 Hash Password. *Journal of Chongqing University of Technology (Natural Science)*, 2013, 7: 014.

Seznam symbolů a zkratek

CUDA	Compute Unified Device Architecture – Architektura pro programování a implementaci na grafických kartách
GPGPU	General-Purpose Computing on Graphics Processing Units - výpočty prováděné pomocí grafických procesorů
CPU	Central Processing Unit - procesor počítače
GPU	Graphic Processing Unit – grafický procesor
RAM	Random Access Memory – operační paměť počítače
API	Application Programming Interface – rozhranní pro programování aplikací
SISD	Single Instruction Single Data – Jedna instrukce a jeden datový proud
SIMD	Single Instruction Multiple Data – Jedna instrukce a více datových proudů
MISD	Multiple Instruction Single Data – Více instrukcí a jeden datový proud
MIMD	Multiple Instruction Multiple Data - Více instrukcí a více datových proudů
SIMT	Single Instruction Multiple Threads - Jedna instrukce pro více vláken
CU	Compute Units – Výpočetní jednotky
PE	Processing Elements – Výpočetní elementy
WI	Work Items - Pracovní jednotky
WG	Work Group – Pracovní skupiny
DLL	Dynamic-Link Library – dynamicky linkovaná knihovna
MB	MegaByte – Jednotka informace
GB	GigaByte – Jednotka informace

Obsah přiloženého DVD

- \\ Diplomová práce \ - složka s elektronickou verzí práce
- \\ Workspace \ - složka se zdrojovými kódy programu
- \\ DLL Knihovna \ - složka s vytvořenou *.dll knihovnou