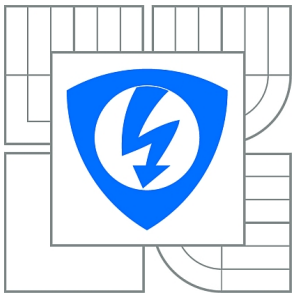
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF CONTROL AND INSTRUMENTATION

PARALELIZACE VÝPOČTŮ POMOCÍ GPGPU PROSTŘEDKŮ

GPGPU PARALLEL COMPUTING

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

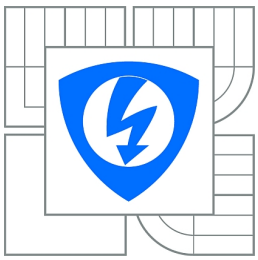
AUTOR PRÁCE — DÁVID PACURA

AUTHOR

VEDOUCÍ PRÁCE — Ing. PETR PETYOVSKÝ

SUPERVISOR

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav automatizace a měřicí techniky

Bakalářská práce

bakalářský studijní obor
Automatizační a měřicí technika

Student:	Dávid Pacura	***ID:***	136569
Ročník:	3	***Akademický rok:***	2012/2013

NÁZEV TÉMATU:

Paralelizace výpočtů pomocí GPGPU prostředků

POKYNY PRO VYPRACOVÁNÍ:

Úkolem studenta je nastudovat a demonstrovat možnosti distribuce výpočtu v PC pomocí grafických procesorů.
1. Seznamte se s paralelními výpočetními prostředky současných grafických karet (GPGPU).
2. Nastudujte možnosti distribuce obecných matematických výpočtů prováděných v grafických procesorech.
3. Definujte a odůvodněte výhody i nevýhody použití GPGPU pro řešení výpočetních úloh.
4. Zvolte vhodný vývojový nástroj pro implementaci algoritmů výpočtu pomocí grafického procesoru.
5. Navrhněte a realizujte několik příkladů demonstrující výpočetní možnosti grafických procesorů.
6. Navrhněte nebo zvolte komplexnější úlohu vhodnou pro paralelizaci pomocí prostředků GPGPU a úlohu vhodně implementujte.
7. Změřte časovou efektivitu výpočtu úlohy při použití GPGPU a běžného CPU. Dále srovnejte náročnost obou implementací.
8. Zhodnoťte dosažené výsledky a navrhněte další možná rozšíření.

DOPORUČENÁ LITERATURA:

[1] Harris, M. : Mapping computational concepts to GPUs; SIGGRAPH 2005

Termín zadání:	11.2.2013	***Termín odevzdání:***	27.5.2013

Vedoucí práce: Ing. Petr Petyovský
Konzultanti bakalářské práce:

doc. Ing. Václav Jirsík, CSc.
Předseda oborové rady

Abstrakt

Cieľom tejto bakalárskej práce je poukázanie na možnosti a demonštrácia paralelizácie výpočtov pomocou grafických procesorov. V práci sú uvedené popisy dostupných vývojových nástrojov, a na ich základe je jeden zvolený na implementáciu šifrovacieho MD5 algoritmu a neurónovej siete na rozpoznávanie čísel. Výkon algoritmov je následne porovnaný so svojím paralelným ekvivalentom pre bežné procesory. V závere sú popísané problémy pri vývoji a spôsoby, ktorými je možné sa im vyhnúť.

Kľúčové slová

GPGPU, GPU, CPU, paralelizácia, OpenCL, CUDA, kernel, MD5, neurónová sieť

Abstract

The aim of this trim's thesis is to reveal possibilities and demonstrate parallelization of computation on graphics processors. The paper presents descriptions of available development tools, and then one of them is selected to implement MD5 encryption algorithm and neural network for optical character recognition. Its performance is then compared to its parallel equivalent for conventional processors. In conclusion, problems encountered during development are described, and ways of avoiding them are discussed.

Keywords:

GPGPU, GPU, CPU, parallelization, OpenCL, CUDA, kernel, MD5, neural network

Bibliografická citace

PACURA, D. Paralelizace výpočtů pomocí GPGPU prostředků. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2013. 60s. Vedoucí bakalářské práce byl Ing. Petr Petyovský.

Prohlášení

„Prohlašuji, že svou bakalářskou práci na téma Paralelizace výpočtů pomocí GPGPU prostředků 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 bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.“

V Brně dne: 27. května 2013 .....

podpis autora

Poďakovanie

Vedúcemu práce Ing. Petrovi Petyovskému za cenné rady a pripomienky.

Ing. Tomášovi Hynčicovi za ochotu a pomoc pri implementácii neurónovej siete.

V Brně dne: 27. května 2013 .....

podpis autora

OBSAH

ZOZNAM OBRÁZKOV

ZOZNAM TABULIEK

ZOZNAM ROVNÍC

1 ÚVOD

Cieľom tejto práce je demonštrácia výkonu algoritmov implementovaných pre grafickú kartu a ich výkonnostného náskoku oproti implementácii pre bežné procesory.

Prvá časť sa bude zaoberať historickým vývojom, ktorý dospel k súčasnej generácii grafických procesorov optimalizovaných pre „bežné“ výpočty. Potom bude pokračovať problémami paralelizácie, dátovými závislosťami a ťažkosťami, s ktorými je možné sa pri písaní paralelných algoritmov stretnúť.

V druhej časti sa prejde priamo ku GPGPU, kde budú rozobrané ich výhody a nevýhody. Potom bude nasledovať rozbor a porovnanie vývojových nástrojov umožňujúcich ich programovanie. Na základe toho bude zvolený jeden z týchto nástrojov, pomocou ktorého budú demonštrované výpočtové možnosti grafických procesorov. Jeho možnosti a princípy budú podrobne rozobrané. Záver tejto časti bude venovaný spôsobom optimalizácie algoritmov implementovaných pre grafické procesory.

V tretej časti bude implementovaných niekoľko algoritmov demonštrujúcich výpočtový výkon grafických kariet a porovnanie s optimalizovanou a paralelizovanou verziou pre procesor. Následne bude zhodnotená náročnosť ich implementácie pre grafické procesory voči implementácii pre CPU. Záver tejto časti sa bude zaoberať dosiahnutým výkonnostným ziskom (zrýchlením) a porovnaním, či tento prekoná náročnosť implementovania algoritmov pre GPU.

V závere budú zhodnotené namerané výsledky a skúsenosti získané pri programovaní grafických procesorov. Taktiež budú navrhnuté potenciálne vylepšenia a smery, ktorými je možné sa uberať v ďalších prácach.

2 ZVYŠOVANIE VÝKONU

S postupnou digitalizáciou sveta okolo nás prichádza aj potreba automatizácie rôznych procesov, dokonalejších simulácií, vyššieho zabezpečenia. Užívatelia vyžadujú realistickejšie virtuálne svety a filmy, vedci sa snažia objaviť lieky na rôzne choroby. Doba, v ktorej žijeme, sa orientuje na technologicky presnejšie, rýchlejšie, komfortnejšie či ekologickejšie dopravné prostriedky. Tento stav si vyžaduje počítače s vysokou efektivitou a výkonom. Uvedené súvislosti sa stali dôvodom neustáleho hľadania nových spôsobov získania extra výkonu. Jedným z nich je využitie grafických procesorov.

2.1 Prechod k multicore a paralelizácii u osobných počítačov

V minulosti postačovalo k zvýšeniu výkonu procesorov zmenšiť výrobný proces, čím bolo umožnené pridať ďalšie tranzistory, a zvýšiť výkon pri zachovaní alebo zmenšení spotreby aj veľkosti jadra. Toto zmenšovanie taktiež umožnilo zvyšovanie frekvencie, čím opätovne došlo k zvýšeniu výkonu. Avšak tento proces narazil začiatkom minulého desaťročia na fyzikálnu bariéru. S vyššou frekvenciou je potrebné aj vyššie napätie, čím neúmerne rastie spotreba procesora a tým aj vyžarované odpadové teplo:

$$P = CU^2 f \qquad (1)$$

kde: *P [W]* je celkový príkon, *C [F]* je kapacita procesora, *U [V]* je jeho napájacie napätie a *f [Hz]* frekvencia na ktorej pracuje.

Túto fyzikálnu limitáciu je možné obísť zvýšením počtu procesorov a znížením ich frekvencie, čím dôjde k poklesu potrebného napätia a zvýšeniu kapacity. Pri zvýšení počtu jadier procesora na dve však k zachovaniu výkonu postačuje zníženie ich frekvencie na polovicu (časť výkonu je samozrejme potrebná k synchronizácii a riadeniu jadier, takže k rovnakému výkonu potrebujem mierne vyššiu frekvenciu). Na základe experimentálnych meraní bolo empiricky stanovené, že každým zdvojnásobením počtu jadier na polovičnej frekvencii narastie kapacita procesora 2,2 krát. Napätie však poklesne na 0,6 násobok pôvodnej hodnoty a spotreba dosahuje iba 0,396 násobok pôvodnej hodnoty [1]. Týmto zároveň dochádza k žiaducemu zvýšeniu efektivity výkon/watt.

2.2 Vývoj grafických kariet

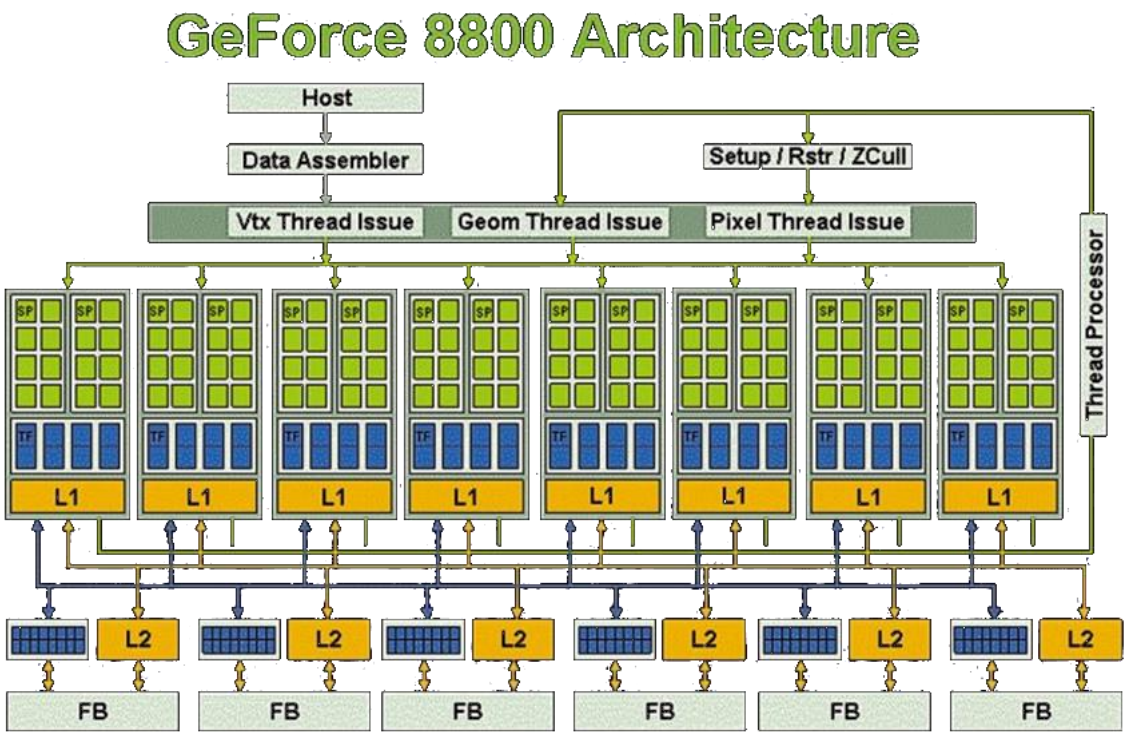
Koncom 80-tych rokov došlo k rozmachu grafických operačných systémov, ktoré práve grafikou výrazne zaťažovali procesor. Ako priamy dôsledok dochádza práve k vzniku grafických kariet - najprv len v podobe 2D akcelerátorov iba za účelom vykresľovať jednoduchú grafiku, neskôr aj ako plnohodnotných 3D grafických kariet. Príchod nového hardwaru viedol tiež k rozšíreniu trhu počítačových hier, ktoré svojou náročnosťou vyžadujú čoraz väčší výkon grafických akcelerátorov. Vznikajú API OpenGL a DirectX, ktoré

ponúkajú nové možnosti výpočtov grafiky, na základe čoho vzniká cyklus: vývojári vyžadujú výkonnejší hardware pre čoraz zložitejšie výpočty, prichádzajú nové verzie API a s nimi výkonnejší a zložitejší hardware.

Začiatkom tretieho tisícročia si niektorí programátori uvedomili, že grafické karty začínajú poskytovať väčší výpočtový výkon ako procesory, a začali ich programovať pomocou grafických, prípadne iných proprietárnych API (Brook, OpenGL) pre bežné výpočty [2]. Tento spôsob však bol výrazne limitujúci a zložitý, preto sa veľmi nepresadil. To sa zmenilo s príchodom poslednej revolúcie u grafických procesorov – zavedením unifikovanej architektúry.

Grafické karty pozostávali prakticky od počiatku z viacerých druhov paralelných koprocesorov (TnL jednotky, pixel a vertex shadery, textúrovacie jednotky...). Nová verzia DirectX 10 z roku 2006 požaduje ďalšie špecializované jednotky. Ich veľkou nevýhodou je, že každá grafická aplikácia ich potrebuje iný pomer a to dokonca v závislosti od vykresľovanej scény. Nepoužité jednotky znamenajú zbytočné plytvanie tranzistormi, preto dochádza k unifikácii týchto jednotiek. Priekopníkom v tejto oblasti je spoločnosť NVIDIA a jej grafické karty rady 8800 s novou architektúrou Tesla z konca roku 2006.

2.2.1 NVIDIA Tesla



Obrázok 1: Architektúra GeForce 8800 [3]

Táto architektúra grafických kariet zavádza nové univerzálne SP jednotky (thread procesory – na Obrázok 1: Architektúra GeForce 8800 zelenou farbou) s jednotnou inštrukčnou sadou, v

počte 8 kusov pre každý blok (karta má 16 blokov, dokopy 128 výpočtových jednotiek). Každá SP podporuje 96 vláken, dokopy maximálne 12288 vlákien (pre porovnanie, súčasné Intel Core i7 CPU podporujú 8-16 vlákien) a obsahuje multiply-add (MAD) jednotku a ďalšiu multiply jednotku, všetko s 32-bitovou presnosťou pri výpočtoch s plávajúcou desatinou čiarkou. Ďalej SFU jednotky (special function units) v počte 2 kusy na blok, ktoré podporujú inštrukcie `rcp` (pre nulu vracia nekonečno, inak prevrátenú hodnotu), `rsqrt` (invertovaná kvadratická odmocnina), `log2`, `exp2`, sin a cos aproximáciu a kvadratickú interpoláciu. Každý blok má 16KB zdieľanej pamäte. Dva takéto bloky majú 8 spoločných textúrovacích jednotiek a L1 cache. Tieto výkonné jednotky majú povolené čítanie a zápis do pamäte aj zdieľanej pamäte. Táto architektúra je navrhnutá tak, aby spĺňala požiadavky IEEE štandardu na jednoduchú presnosť výpočtov s plávajúcou desatinou čiarkou a umožňovala vysokú škálovateľnosť. Jej inštrukčná sada je prispôsobená nielen pre spracovanie grafiky, ale aj pre všeobecné výpočty. Zároveň s uvedením tejto architektúry dochádza k uvedeniu CUDA (Compute Unified Device Architecture), ktorá pomocou nadstavby jazyka C umožňuje programátorom obsluhovať GPU. Tým sa dostáva do širšieho povedomia odvetvie GPGPU (General-purpose computing on graphics processing units), ktoré bolo do tejto doby veľmi okrajovou záležitosťou pre pár nadšencov. GPU bolo v tej dobe nutné programovať pomocou grafických API OpenGL alebo DirectX, čím vznikala nutnosť prepísať riešený problém zo štandardného algoritmu na grafický problém. Väčšinou sa využíval Pixel Shader, ktorý pre určité súradnice pixela [x,y] na základe ďalších parametrov počítal finálnu hodnotu farby pre daný pixel. Vývojári si uvedomili, že ako vstupné parametre môžu použiť prakticky akékoľvek dáta. Následne došlo k určitému výpočtu a po jeho dokončení sa namiesto vykreslenia pixelov na obrazovku hodnoty odoslali späť na CPU. Tieto implementácie však nepodporovali výpočty s plávajúcou desatinou čiarkou, čím sa značne znižoval okruh riešiteľných úloh. Nevýhodou tejto technológie je jej proprietárnosť (funguje iba na GPU od NVIDIE). Na uvedené udalosti reagovala spoločnosť Apple vytvorením vlastného API OpenCL. Toto bolo následne ďalej vyvíjané a štandardizované v spolupráci s firmami IBM, Intel, AMD a NVIDIA. V závere roka 2008 vzniká jeho prvá špecifikácia. NVIDIA následne implementuje celú špecifikáciu OpenCL do svojho proprietárneho frameworku CUDA a pridáva ďalšie rozšírenia.

3 PARALELIZÁCIA

Paralelizácia výpočtov znamená, že výpočty sú prevádzané zároveň (v tom istom čase). Tým dochádza k skráteniu celkového času potrebného pre výpočet. S týmto prístupom však prichádzajú aj rôzne nevýhody, hlavne ďalšie druhy návrhových (programátorských a implementačných) chýb, nevyskytujúcich sa pri jednovláknových aplikáciách, napríklad špecifická synchronizácia vláken. Ďalšou nevýhodou je, že paralelné spracovanie dát musí byť z veľkej časti implementované ručne, programátorom, čo zaberie množstvo času a prostriedkov. V súčasnosti už existujú nástroje (kompilátory), ktoré sú z časti schopné optimalizovať sériový kód na paralelný, avšak majú veľké množstvo obmedzení a ručne paralelizovaný kód je takmer vždy výkonnejší ako ten automaticky generovaný [4].

3.1.1 Automatická paralelizácia

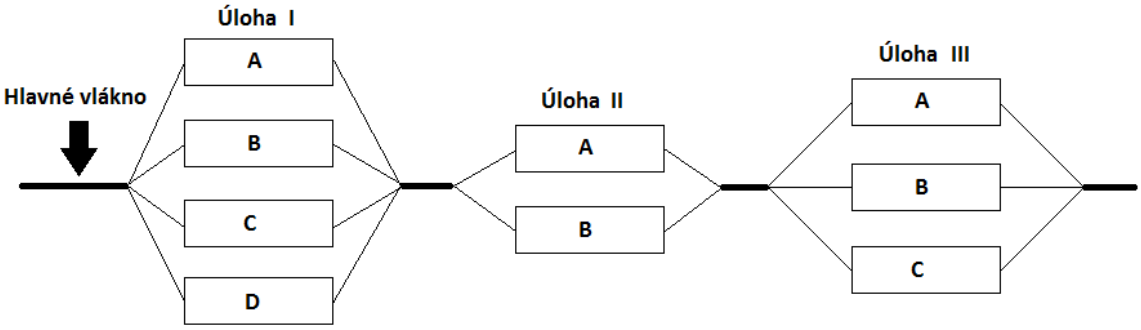
Automatická paralelizácia v súčasnosti vyžaduje pred akýmikoľvek úpravami kódu komplexnú analýzu programu za účelom určenia častí, ktoré je možné paralelizovať a následné vyhodnotenie, či ich paralelizácia bude prínosná (aby nedošlo k situácii, kedy samotné rozdelenie úlohy medzi procesory zaberie viac času ako sériový výpočet). V súčasnosti však táto analýza nie je dostatočne inteligentná na to, aby pochopila, čo daný algoritmus vykonáva, prípadne navrhla vhodnú alternatívu, ktorá bude výkonnejšia a/alebo vhodnejšia na paralelizáciu. Hľadá preto iba určité vzory, o ktorých vie, ako ich spracovať. Ďalšou požiadavkou je známy rozsah vstupných dát v čase kompilácie, čo však u väčšiny programov nie je dopredu známe. Z uvedených obmedzení zostáva plne automatická paralelizácia kompilátorom, aj napriek desaťročiam vývoja, pre bežné programovacie jazyky stále v nedohľadne.

3.1.2 Poloautomatická paralelizácia

Poloautomatická paralelizácia je v súčasnosti najpoužívanejšia forma paralelizácie výpočtov. Jej princíp spočíva v označení bloku (na základe použitej knižnice), ktorý sa má paralelizovať, programátorom. Ten sa už následne nemusí zaoberať tým, ako bude úloha rozdelená medzi procesory, ani tým, koľko vlákien je možné spustiť. Jeho jedinou úlohou je overiť, či v danom bloku neexistuje závislosť medzi dátami, ktorá znemožňuje paralelizáciu.

3.1.3 Nástroje umožňujúce paralelizáciu

Väčšina nástrojov umožňujúcich automatickú paralelizáciu využíva k prevodu sériového kódu na paralelný knižnicu OpenMP, ktorú spoločne vyvíjajú poprední svetoví softwaroví a hardwaroví producenti. Tá využíva jedno hlavné vlákno, ktoré je zodpovedné za vytvorenie potrebného počtu podradených vlákien a rozdeleniu úloh medzi ne. Samotné hlavné vlákno taktiež vykonáva úlohy [4] (Obrázok 2).



Obrázok 2: Princíp činnosti vláken v knižnici OpenMP

Tieto automatické nástroje sú najefektívnejšie pri malých, nie príliš komplikovaných cykloch. Ich nevýhodou je, že existuje možnosť vyhodnotenia cyklu ako vhodného na paralelizáciu, ktorý v konečnom dôsledku prinesie spomalenie programu kvôli réžii okolo vytvárania, synchronizácie a ukončenia vláken.

Niektoré z nástrojov umožňujúcich automatickú paralelizáciu, väčšina z nich využíva práve vyššie uvedenú knižnicu OpenMP:

- Intel C++ Compiler
- iPat/OMP
- SUIF Compiler
- Omni OpenMP Compiler
- S2P
- Par4All

V prípade manuálnej paralelizácie existuje viacero spôsobov. Jednou z možností je využitie knižníc ako OpenMP (programátor napríklad špeciálnym príkazom označí cyklus, zvyšok zabezpečí daná knižnica), ktorá je súčasťou Microsoft Visual Studio 2008 a vyšších. Ďalšou možnosťou je písanie programu v jazyku, ktorý priamo podporuje paralelizáciu (napríklad Parallel C).

3.2 Amdahlov a Gustafsonov zákon

V ideálnom prípade zrýchlenie dané paralelizáciou by malo byť lineárne – zdvojnásobením počtu spracovávajúcich prvkov by sa mala doba behu programu skrátiť na polovicu a opätovným zdvojnásobením druhýkrát znova na polovicu. Avšak iba veľmi málo algoritmov dosahuje optimálne zrýchlenie. Veľa z nich má takmer lineárne zrýchlenie u malého počtu spracovávajúcich prvkov, čo vyúsťuje v konštantnú hodnotu pri veľkých počtoch spracovávajúcich prvkov.

Potenciálne zrýchlenie algoritmu pri platforme paralelných výpočtov je dané Amdahlovým zákonom, pôvodne sformulovaným Genom Amdahlom v roku 1960 [5]. Tento zákon hovorí, že malá časť programu, ktorá nemôže byť paralelizovaná, bude limitovať celkové zrýchlenie umožnené paralelizáciou. Akékoľvek matematické alebo inžinierske problémy budú typicky pozostávať z niekoľkých paralelizovateľných a neparalelizovateľných častí. Tento vzťah je daný rovnicou :

$$\mathrm{S(N)} = \frac{1}{(1 - \mathrm{P}) + \frac{P}{N}} \tag{2}$$

kde: S je zrýchlenie programu (faktor pôvodnej sekvenčnej dĺžky behu), P je pomerná časť programu, ktorá je paralelizovateľná (ak $P=1$, cely program je paralelizovateľný) a N je počet procesorov použitých na paralelný výpočet. To znamená, že ak sekvenčná časť programu je 10% dlhá, P bude rovné 0,9. Z toho vyplýva, že nemôžeme dostať viac ako 10-násobne zrýchlenie, bez ohľadu na to, koľko procesorov sme pridali a pridávanie ďalších procesorov výrazne znižuje pomer výkon/cena. Tieto súvislosti udávajú horný limit užitočnosti pridávania ďalších paralelných jednotiek.

Predpokladajme, že úloha pozostáva z dvoch nezávislých častí A a B. A zaberá 20% potrebného času na celý výpočet. S úsilím by programátor mal byť schopný zrýchliť túto časť 5x. Oproti tomu, na dvojnásobné zrýchlenie časti B bude potrebné menšie úsilie. Výpočtom podľa vzorca (3) zistíme, iba že 5-násobným zrýchlením časti A dosiahneme maximálne zrýchlenie približne 1,19 násobku pôvodného času, avšak dvojnásobné zrýchlenie iba časti B prinesie zrýchlenie na približne 1,67 násobok pôvodného času.

$$S_m \leq \frac{p}{1 + t(p - 1)} \tag{3}$$

kde: S_m je maximálne zrýchlenie, p je násobok zrýchlenia, a t časť času, ktorú neparalelizujeme.

Gustafsonov zákon [6] je ďalší zákon v počítačovom inžinierstve, blízko príbuzný Amdahlovmu zákonu. Na rozdiel od Amdahlovho, zohľadňuje celkový výkon, ktorý získame pridaním procesorov, čo umožňuje vyriešiť väčší problém v rovnakom čase.

Amdahlov zákon predpokladá, že veľkosť vyriešeného problému a veľkosť sekvenčnej časti je nezávislá na počte procesorov, zatiaľ čo Gustafsonov predpokladá, že so zvýšením počtu procesorov dôjde aj k nárastu problému (hlavne paralelizovateľnej časti), ktorý bude vyriešený v približne rovnakom alebo kratšom čase.

3.3 Dátové závislosti

Porozumenie dátovej závislosti je základ implementácie paralelných algoritmov. Žiaden program nemôže bežať rýchlejšie ako najdlhšia časť reťazca v závislých kalkuláciách, pokiaľ kalkulácie, ktoré závisia na predchádzajúcich výpočtoch v reťazci musia byť vykonané v poradí. Avšak veľa algoritmov nepozostáva iba z reťazca závislých kalkulácií, zvyčajne tu sú možnosti vykonať kalkulácie paralelne, prípadne je možné tieto závislosti odstrániť úpravou algoritmu.

3.3.1 Bernsteinove podmienky

Bernsteinove podmienky [7] sú súborom pravidiel na zistenie, či dve úlohy môžu byť vykonané paralelne. Povedzme, že P_i a P_j sú dve časti programu. Definujme I ako všetky vstupné premenné a O ako všetky výstupné premenné. P_i a P_j sú nezávislé, ak spĺňajú:

$$I_i \cap O_j = \emptyset \quad (4)$$

$$I_j \cap O_i = \emptyset \quad (5)$$

$$O_i \cap O_j = \emptyset \quad (6)$$

kde ∩ je prienik a ∅ prázdna množina.

Z rovnice (4) vyplýva, že vstupné premenné P_i nesmú byť súčasťou P_j, z rovnice (5), že vstupné premenné P_j nesmú byť súčasťou P_i, z rovnice (6), že výstupy oboch procesov musia byť taktiež rozdielne. Ak sú splnené všetky podmienky, tieto procesy môžu byť vykonané zároveň.

Uvažujme funkcie, ktoré demonštrujú niekoľko druhov závislostí:

Tabuľka 1: Neparalelizovateľná funkcia

```
1:   function Dep (a,b)
2:       c:=a*b;
3:       d:=2*c;
4:   end function
```

Podľa Tabuľka 1 operácia 3 nemôže byť vykonaná pred (alebo zároveň s) operáciou 2, pretože operácia 3 používa výsledok operácie 2.

Tabuľka 2: Paralelizovateľná funkcia

```
1:   function NoDep (a,b)
2:       c:=a*b;
3:       d:=2*b;
4:       e:=a+b;
5:   end function
```

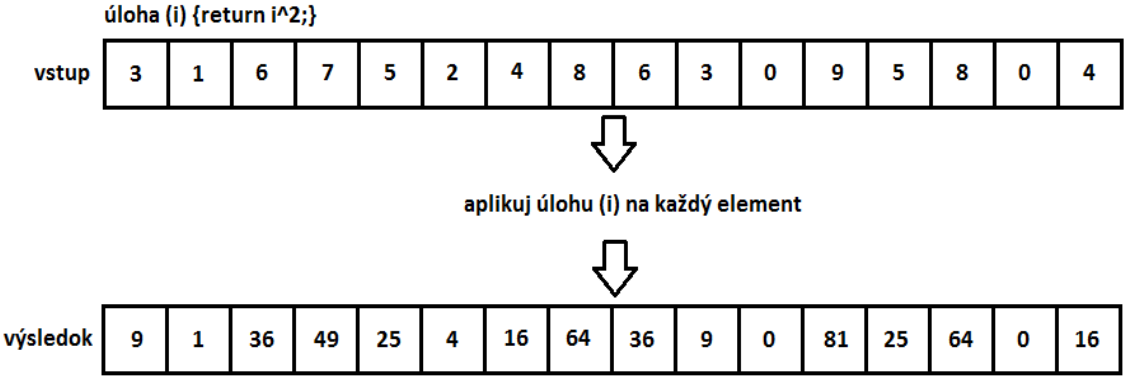
V príklade z Tabuľka 2 nie sú závislosti medzi príkazmi, preto všetky môžu bežať paralelne (je možné využiť tri vlákna).

Z príkladov vyplýva, že v niektorých situáciách je možné algoritmus prepísať tak, aby ho bolo možné spravovať paralelne. Táto úprava si môže vyžiadať dlhší kód, respektíve môže byť potrebný dlhší čas na počiatočnú inicializáciu, ale pri dostatočnom objeme dát bude v konečnom dôsledku výsledná rýchlosť spracovania vyššia.

3.4 Programovacie modely

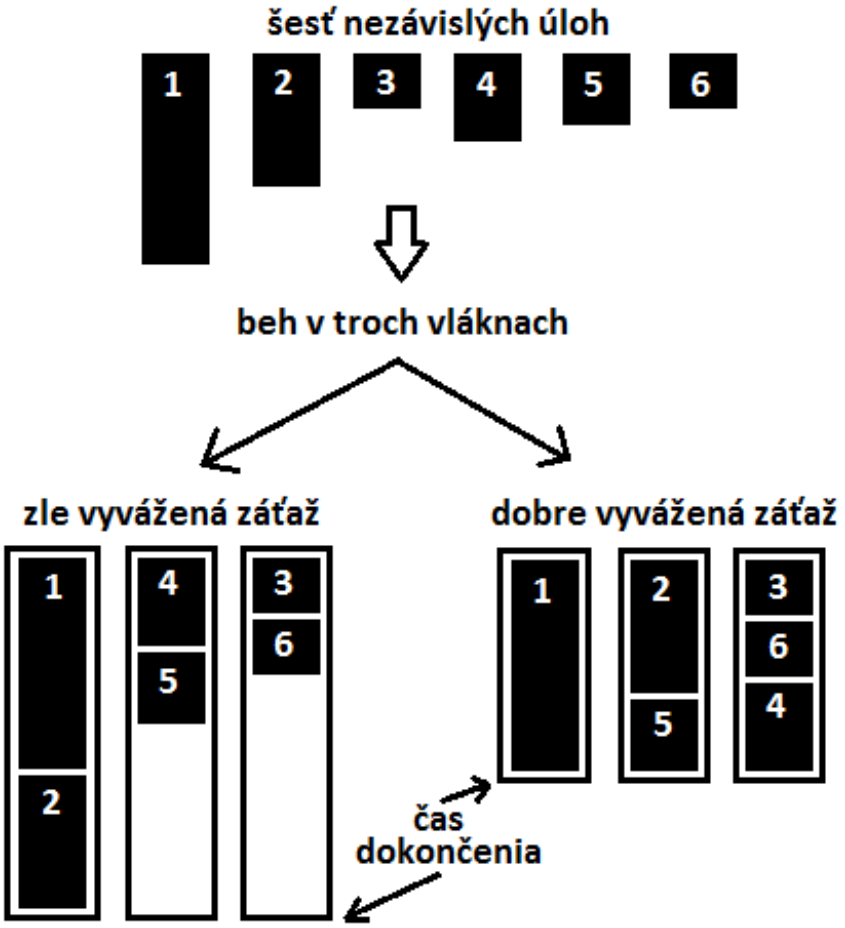
Existujú dva hlavné programovacie modely: úlohový paralelizmus a dátový paralelizmus.

V dátovom paralelizme môžu byť všetky dáta zmenené v jednom momente. Paralelizmus je vyjadrený aplikovaním určitej sady inštrukcií na všetky dáta zároveň.



Obrázok 3: Jednoduchý príklad dátového paralelizmu, kde je jedna úloha aplikovaná na všetky dáta zároveň

Naproti tomu, v úlohovom paralelizme sú na dáta aplikované rôzne inštrukcie. Celkové dokončenie výpočtu vyžaduje dokončenie všetkých čiastočných úloh. Tu vzniká problém vyváženia záťaže. Keďže na dokončenie rôznych úloh je potrebný rozdielny čas, je vhodné tieto úlohy priradiť výpočtovým jadrám nie náhodne, ale s ohľadom na dĺžku ich trvania tak, aby výpočet na všetkých jadrách končil v približne rovnakom čase. Inak by mohlo dôjsť k výraznému spomaleniu výpočtu, čím by sme prišli o výhodu paralelného spracovania (program by bol samozrejme stále rýchlejší ako sériový, avšak nebol by využitý plný potenciál paralelizácie).



Obrázok 4: Úlohový paralelizmus ukazujúci dve možnosti pridelenia úloh

Súčasné grafické karty podporujú obidva druhy paralelizmu, avšak úlohový paralelizmus je limitovaný. Obmedzenie spočíva v nemožnosti spustiť na každej výpočtovej jednotke grafickej karty iný program (toto už neplatí pre poslednú generáciu grafických kariet NVIDIA). Dátový paralelizmus je taktiež výrazne rýchlejšie (pri vetvení sa v skutočnosti vykonajú všetky vetvy, čím dochádza k spomaleniu). Tým súčasné GPU spadajú do kategórie SIMD (Single instruction, multiple data) paralelných procesorov (takzvané vektorové procesory).

4 VYUŽITE GPGPU

V súčasnosti sa grafické karty využívajú v rôznych odvetviach priemyslu a výskumu. Častokrát umožnili spracovávať dáta novým spôsobom, čím výrazne urýchlili simulácie, prípadne matematické výpočty. Najväčší prínos je v nasledujúcich aplikáciách [8]:

- Spracovanie audio a videosignálov (analýza, konverzia, redukcia šumu, vylepšenie)
- Renderovanie realistickej počítačovej grafiky
- Simulácia fyzikálnych procesov (kolízie objektov, seizmické procesy)
- Spracovanie dát magnetickej rezonancie v reálnom čase
- Kryptografia (šifrovanie / dešifrovanie hesiel)
- Distribuované výpočty (hľadanie vhodných liekov na rôzne choroby, analýza genómu)

5 VÝHODY A NEVÝHODY GPGPU

Výhoda grafických kariet nad procesormi spočíva v obrovskom počte relatívne pomalých jadier, avšak iba v prípade, že danú výpočtovú úlohu je možné dostatočne paralelizovať a je typu SIMD. Vtedy je typicky možné dosiahnuť 100-násobné zrýchlenie, v prípade výskytu vetvenia, úlohového paralelizmu alebo nedostatočnej miery paralelizácie je bežné zrýchlenie 10x [9]. Samozrejme, je ťažké presne definovať ekvivalentné GPU pre dané CPU. Asi najvhodnejším parametrom ekvivalencie je v prípade GPGPU rovnaká cenová hladina v čase nákupu hardwaru. Ďalšou výhodou je vysoká rýchlosť prenosu dát medzi samotným GPU a jeho dynamickou pamäťou (DRAM). U GeForce 8800 to bolo 85 GB/s, u GeForce 580 je to už 192 GB/s, čo je približne 10-násobne viac ako u CPU (pre porovnanie, Intel Core 2 procesory majú prenos približne 8 GB/s a u dnes najvýkonnejších Core i5 a i7 procesorov je to zhruba 20 GB/s).

Grafické karty sú však stále iba koprocesormi – vyžadujú CPU na prideľovanie úloh a prísun dát k výpočtu.

Medzi hlavné nevýhody patrí nedostupnosť týchto funkcií:

- Virtuálna pamäť
- Dynamická alokácia pamäte na zariadení za behu programu
- Adresácia iných zariadení ako pamäť (klávesnica, tlačiareň)
- Pointre na funkcie (nepriame volania) – dostupné v CUDA od NVIDIA Fermi architektúry

Táto nedostupnosť bráni vytvoreniu implementácie moderného operačného systému, schopného bežať na GPU.

Ďalšími nevýhodami je dlhá odozva pri prenose dát medzi CPU a GPU a limitované dátové štruktúry (napr. problematická implementácia viazaného zoznamu). Použitie dvojitej presnosti pri výpočte výrazne znižuje výkon (približne na polovicu), preto je nutné zvážiť jej použitie vo všetkých častiach programu. Avšak väčšinu týchto obmedzení kompenzuje výborný pomer cena/výkon pri špecifických úlohách.

Podľa firmy NVIDIA, ktorá na GPU Technology Conference (konala sa 18. – 21. marca 2013 v San Jose, Kalifornia, USA) ukázala plánované generácie grafických kariet a technológie ktoré majú podporovať, však budú tieto nedostatky v blízkej budúcnosti odstránené (generácia Maxwell má v roku 2014 priniesť podporu unifikovanej virtuálnej pamäte umožňujúcou prístup k I/O zariadeniam).

Pre dostatočný prínos GPGPU je potrebný veľký objem dát na spracovanie, inak dôjde k situácii, kedy réžia na prenos dát zaberie viac času ako samotný výpočet. CPU je potom rýchlejšie aj napriek sekvenčnému, prípadne semisekvenčnému spracovaniu dát.

6 VÝVOJOVÉ NÁSTROJE PRE GPGPU

V praxi je možné sa stretnúť s tromi nástrojmi na implementáciu algoritmov na GPU. Dva z nich sú proprietárnymi riešeniami. Ich základná implementácia je v princípe rovnaká a sú z časti kompatibilné (dané hardwarom, na ktorom bežia). Všetky tieto riešenia podporujú synchronizáciu vlákien a atomické operácie (pri zápise všetkých vlákien do jednej zdieľanej premennej je nutné zabezpečiť, aby v jednom okamihu zapisovalo iba jedno vlákno, inak môže dôjsť k nekonzistencii dát na výstupe). Taktiež sa vyžaduje grafická karta s hardwarovou podporou DirectX 10 a novším. (NVIDIA rada 8000 a novšia, ATI rada 2000 a novšia)

6.1 CUDA

Ide o proprietárnu technológiu od firmy NVIDIA uvedenú s príchodom unifikovanej architektúry grafických kariet v Novembri 2006. Hlavným obmedzením je použiteľnosť iba na hardware od firmy NVIDIA. Výhodou je podpora najpoužívanejších operačných systémov (Windows, Linux a Mac OS X) a jednoduchosť úpravy programu na štandard OpenCL, kedy stačí iba prepísať kľúčové slová. Ďalej sú dostupné rozsiahle knižnice riešených problémov a výrazná podpora pre vývojárov. Priebežne sú vydávané nové minoritné verzie, s každým vydaním architektúry grafických kariet aj majoritné, ktoré prinášajú výrazné zmeny a urýchlenia, poväčšine použiteľné iba na týchto nových kartách.

Štandardne sú podporované programovacie jazyky C/C++ a Fortran s CUDA-špecifickými rozšíreniami na vyjadrenie paralelizmu, kooperácie vlákien a podobne. Taktiež existujú riešenia tretích strán, umožňujúce programovanie v širokej škále programovacích jazykov, ako Python, C#, Perl, Java, Haskell, MATLAB a iné.

NVIDIA zaručuje, že všetky programy vytvorené na starších verziách CUDA, budú funkčné na všetkých budúcich grafických kartách.

6.2 OpenCL

Táto technológia je otvoreným štandardom vytvoreným spoločnosťou Apple a v súčasnosti je spravovaná spoločnosťou Khronos Group. Bola uvedená v novembri 2008 ako reakcia na CUDA. Jej najväčšou výhodou je hardwarová a softwarová multiplatformnosť. Je to otvorený priemyslový štandard pre paralelné programovanie heterogénnych počítačových systémov. Okrem klasických grafických kariet môžu programy napísané podľa tohto štandardu bežať prakticky na každom druhu procesorov vrátane mobilných zariadení, prípadne aj na embedded systémoch. Jedinou podmienkou je nutnosť existencie OpenCL ovládačov a kompilátora pre danú architektúru. Nevýhodou je problematické ladenie kódu, kedy je najvýhodnejšie portovať už fungujúci kód pre CPU.

6.3 DirectCompute

Ide o proprietárnu technológiu od firmy Microsoft uvedenú s príchodom DirectX 11 API v októbri roku 2009. Je primárne určená na realistickejšiu fyziku a obraz v herných enginoch. Nevýhodou je obmedzenie na operačné systémy Windows Vista a novšie a nutnosť používať programovací jazyk podobný HLSL (High Level Shader Language). Tým sa prakticky diskvalifikuje zo širokého rozšírenia a ostane iba ako náhrada za predchádzajúce technológie pre herných vývojárov, ktorým bude dané API, na rozdiel od ostatných technológií, relatívne známe. DirectCompute umožňuje efektívnu spoluprácu s prostriedkami DirextX a relatívne jednoduchú integráciu do herných enginov.

7 VOĽBA NÁSTROJA

Na základe informácií uvedených v kapitole 6: Vývojové nástroje pre GPGPU bol ako vhodný vývojový nástroj zvolený OpenCL, keďže je multiplatformový a je možné porovnať reálnu výkonnosť grafických kariet majoritných výrobcov AMD a NVIDIA oproti nimi udávanému teoretickému výkonu.

7.1 Základné princípy OpenCL

Model OpenCL v princípe pozostáva z dvoch vecí:

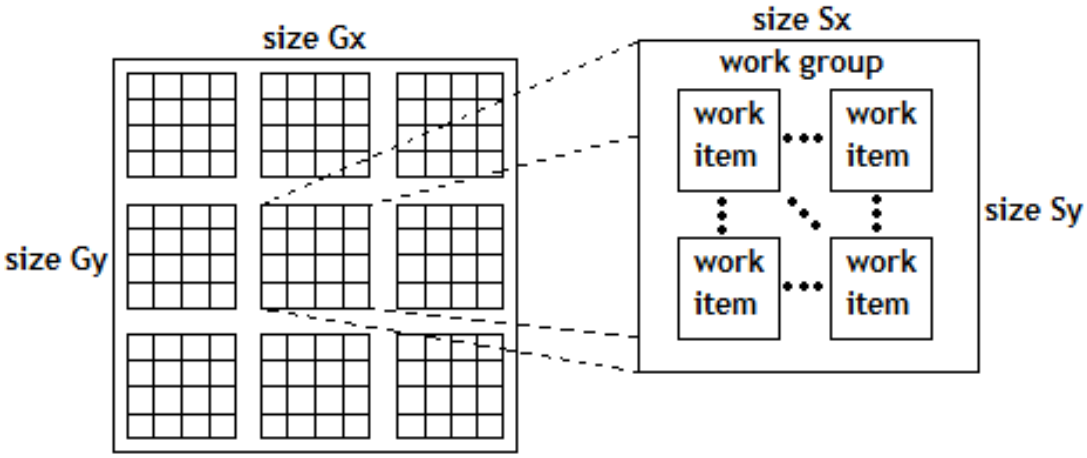
- Jazyka podobného štandardu C99, nazýva sa OpenCL C. Je navrhnutý tak, aby bol jednoduchý na skompilovanie pre GPU. Oproti štandardu má určité obmedzenia a obsahuje navyše nové kľúčové slová a dátové typy.
- Knižnice navrhnutej na kompiláciu OpenCL C kódu a jeho beh

Pre správne pochopenie konceptu OpenCL je najprv nutné vysvetliť nasledujúce pojmy:

- **Platforma** – Je to set zariadení určitého výrobcu umožňujúci spustenie programu napísaného v OpenCL. Počítačový systém môže obsahovať viacero takýchto platforiem.
- **Zariadenia** – Sú obsiahnuté v platforme. Reprezentujú fyzické zariadenia v danom počítačovom systéme schopné spustiť program napísaný v OpenCL. Zariadením môže byť CPU, GPU alebo iný špecializovaný hardware.
- **Hostiteľ** – Názov časti aplikácie, ktorá beží výhradne na CPU. V princípe je to „bežný" počítačový program. Označuje sa ako „hostiteľský kód".
- **Kontext** – Set zariadení, ktoré sa budú používať v programe. Vyžaduje sa, aby boli použité iba zariadenia z vybranej platformy. Umožňuje zdieľanie prostriedkov ako sú pamäť, fronty a zásobníky, medzi zariadeniami, ktoré obsahuje.
- **Kernel** – Názov pre funkciu, ktorá bude vykonaná paralelne na zariadení. Slúži ako vstupná funkcia programu, ako jediná môže byť volaná hostiteľom. Označuje sa kľúčovým slovom `__kernel` a návratová hodnota musí byť typu `void`. Môže volať ďalšie funkcie v programe. Za normálnych okolností sú kompilované až pri spustení, čo zabezpečuje hostiteľ. Označuje sa ako „kód zariadenia"
- **Pracovná položka** (Work-item) – Najmenšia exekučná jednotka. V princípe sa jedná o ekvivalent vlákna, beží na zariadení. Každá inštancia (vlákno) dostane priradené jedinečné identifikačné číslo, na základe ktorého je možné identifikovať dáta, s ktorými má daná pracovná položka pracovať.
- **Pracovná skupina** (Work-group) – Skupina pracovných položiek, ktorá umožňuje spoluprácu medzi nimi. Taktiež má svoje vlastné unikátne identifikačné číslo. Pracovné

položky sú v skupine organizované ako N-dimenzionálne pole, kde N nadobúda hodnoty 1, 2 alebo 3.

- **ND-Range** – Skupina organizujúca pracovné skupiny. Opäť ako N dimenzionálne pole s 1, 2 alebo 3 rozmermi.



Obrázok 5: Príklad ND-Range organizácie [10]

OpenCL kód určený pre zariadenie je kompilovaný až pri spustení a to z dôvodu, že pre každé zariadenie je potrebný iný kompilátor, nakoľko zariadenia môžu byť rôznorodých architektúr a s podporou rôznych inštrukcií. Jeho počiatočná inicializácia je preto pomalšia. Existuje však možnosť uložiť program po prvom skompilovaní v užívateľskom systéme.

V skutočnosti zariadenie nie je schopné spracovať nekonečné množstvo vláken zároveň. OpenCL preto nadbytočné množstvo automaticky zaradí do fronty, z ktorej bude v prípade dokončenia niektorej z úloh čerpať. Programátor v tomto prípade nemá nad poradím vykonávania týchto úloh žiadnu kontrolu.

Vo všeobecnosti platí, že vlákna nebudú zapisovať na tie isté miesta v pamäti, avšak nie je možné sa na to spoľahnúť. Je to „daň“ za vysokú rýchlosť spracovania paralelného kódu. Programátor by sa mal vyvarovať napríklad inkrementácii rovnakého miesta v pamäti z rôznych vlákien. Uvažujme o premennej `count=0`. Typickým príkladom je použitie príkazu `count++`, kedy prvé vlákno prečíta hodnotu 0, pridá k nej 1, avšak medzitým niektoré s ostatných vlákien prečítajú z premennej hodnotu 0 a zapíšu späť tú istú hodnotu (1) ako prvé vlákno. Tým vznikajú takzvané race conditions. Takýto kód môže viesť k nesprávnym výsledkom. Taktiež exituje možnosť použiť atomické operácie, ktoré daný problém odstraňujú, avšak výrazne znižujú výkon.

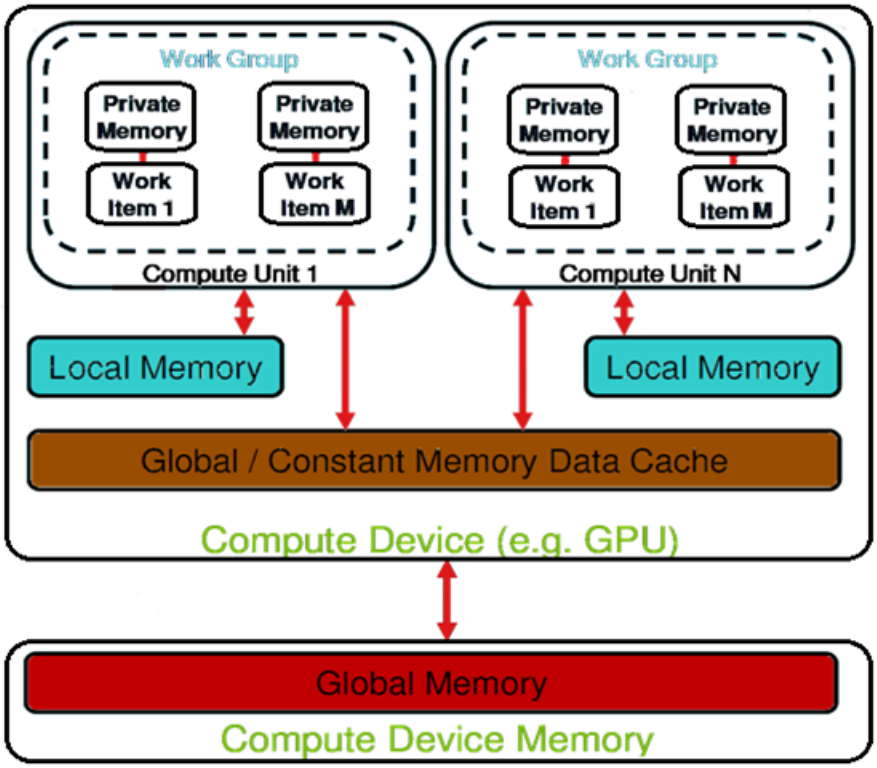
Už zo samotného multiplatformového zamerania OpenCL vyplýva, že kernel pravdepodobne nebude schopný špičkového výkonu na všetkých druhoch zariadení [10].

7.2 Pamäťové priestory

Každé zariadenie má 4 druhy pamäťových priestorov, ktoré majú rôznu časovú odozvu a veľkosť a na ich prečítanie/zápis je potrebný rozdielny počet cyklov. Označujú sa kľúčovým slovom, pre každý priestor existujú určité reštrikcie. V prípade neoznačenia priestoru premennej je táto implicitne alokovaná v privátnej časti.

Štandard OpenCL definuje nasledujúce pamäťové priestory:

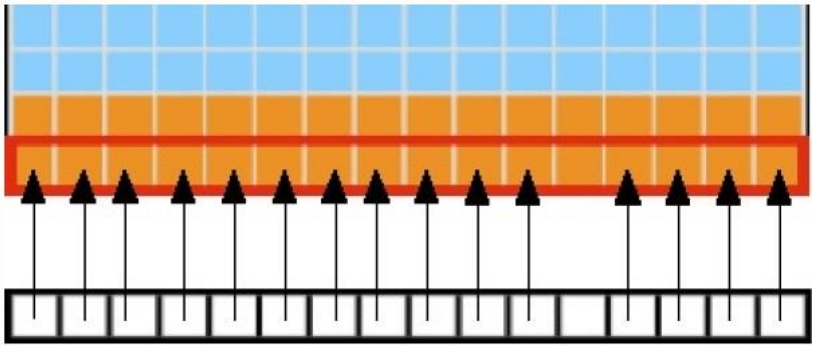
- **Globálna** (`__global`) – RAM pamäť zariadenia, prístup je časovo najnáročnejší, zároveň jediná pamäť, do ktorej môže pristupovať aj hostiteľ
- **Konštantná** (`__constant`) – globálna pamäť určená iba na čítanie, na určitých zariadeniach je načítaná do pamäte na čipe (cached) pre zrýchlenie prístupu
- **Lokálna** (`__local`) – rýchla oblasť pamäte (približne 100x oproti globálnej) spoločná pre pracovnú skupinu, umožňuje kooperáciu vlákien v danej skupine. Je vhodná ako pomocná pamäť na obmedzenie prístupu k pamäti globálnej.
- **Privátna** (`__private`) – súkromná pamäť každej pracovnej položky, pokiaľ možno uložená v cache na čipe, po vyčerpaní dostupných prostriedkov uložená v globálnej pamäti



Obrázok 6: Pamäťové priestory OpenCL [11]

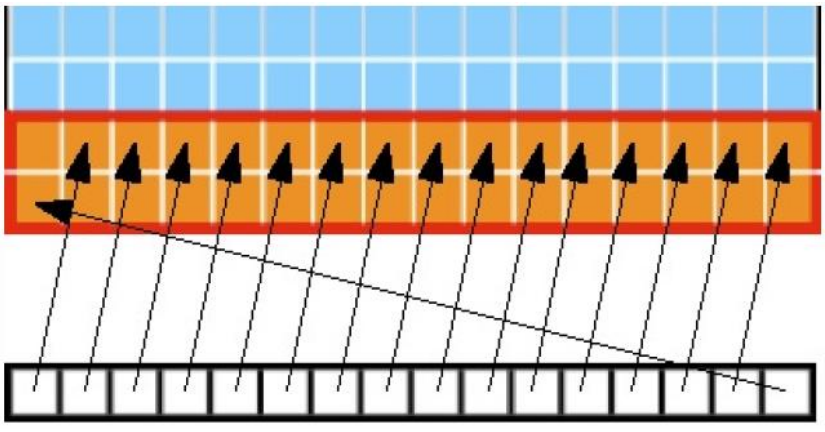
7.3 Prístup do globálnej pamäte GPU

Z výkonového hľadiska je vhodné, aby prístup do pamäte bol splývajúci, to znamená, že k-te vlákno pristupuje do k-teho prvku v poli (Obrázok 7: Splývajúci prístup do pamäte, všetky vlákna okrem jedného pristupujú k svojmu korešpondujúcemu slovu). Týmto sa zabezpečí maximálny výkon, keďže všetky dáta sa nachádzajú v jednom segmente a je možné ich načítať naraz.

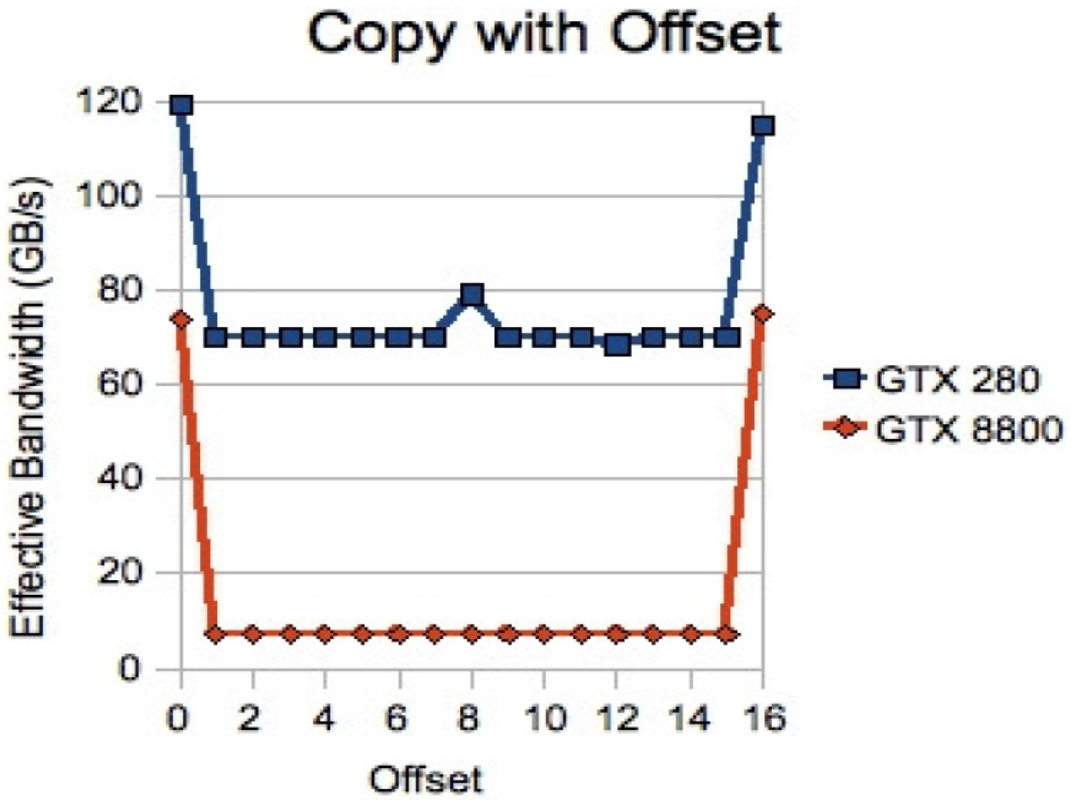


Obrázok 7: Splývajúci prístup do pamäte, všetky vlákna okrem jedného pristupujú k svojmu korešpondujúcemu slovu [12]

V prípade, že k-te vlákno pristupuje do (k+n)-tého prvku (použitie offsetu, Obrázok 8: Prístup do pamäte s hodnotou offset = 1), ide o posunutý prístup. V tomto prípade dochádza k výkonnostnému prepadu, keďže na prečítanie všetkých hodnôt sú už potrebné dva prístupy do pamäte. Výkonnostný dopad v závislosti na veľkosti hodnoty offset zobrazuje Obrázok 9: Výkon kopírovania dát s rozdielnym offsetom. Na grafe je možné vidieť, že v prípade GTX 280 je prepad výkonu takmer na polovicu, keďže sú potrebné dva prístupy do pamäte namiesto jedného. V prípade staršej GTX 8800 je však v prípade použitia offsetu, ktorý nie je násobkom čísla 16, prepad takmer 8-násobný, čo je spôsobené nutnosťou 16 prístupov do pamäte.

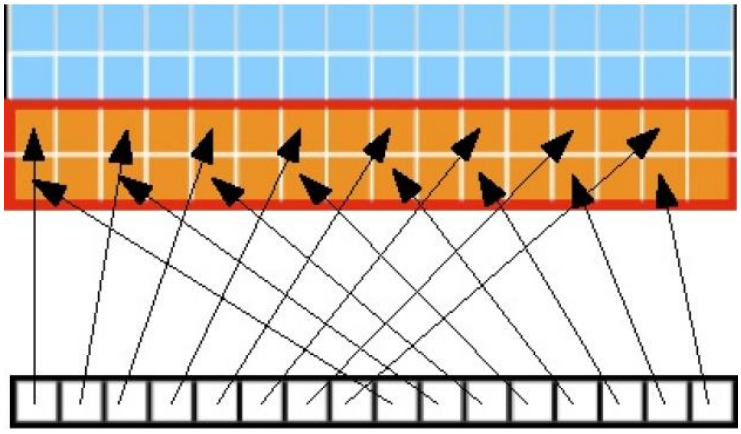


Obrázok 8: Prístup do pamäte s hodnotou offset = 1 [12]



Obrázok 9: Výkon kopírovania dát s rozdielnym offsetom [12]

Ďalším spôsobom prístupu do pamäte je prístup s rôznym krokom (Obrázok 10: Prístup do pamäte s krokom 2). So zvyšujúcim sa krokom rastie aj počet potrebných prístupov do pamäte, až kým nedosiahne počet vlákien v jednom bloku. Typickým príkladom je prístup po stĺpcoch k matici uloženej po riadkoch.



Obrázok 10: Prístup do pamäte s krokom 2 [12]

V prípade, že aspoň časť vlákien pristupuje k rovnakému miestu v globálnej pamäti, dôjde iba k jednému čítaniu a bude vykonaný broadcast pre vlákna čakajúce na danú hodnotu. Týmto dochádza k zefektívneniu prístupu a skráteniu času potrebného na načítanie dát [12].

7.4 Konflikty lokálnej pamäte GPU

Lokálna pamäť sa skladá z buniek, kde sú za sebou nasledujúce 32-bitové slová pridelené nasledujúcim bunkám. Zápis/čítanie rôznych buniek prebieha paralelne, avšak v prípade, že sa požaduje prístup k jednej bunke viackrát, dochádza k serializácii prístupu a tým k spomaleniu.

7.5 Základné možnosti optimalizácie programov pre GPU

- Počet pracovných jednotiek by mal byť násobkom čísla 32 (NVIDIA) alebo 64 (AMD), čo je veľkosť jedného bloku
- Na zariadení môže bežať iba jeden kernel v danom čase (neplatí pre poslednú generáciu kariet NVIDIA)
- Dostatočný počet blokov môže skryť odozvu prístupu do pamäti (v prípade, že jeden blok vlákien čaká na dáta z pamäti, iný môže medzičasom využiť výpočtové jednotky)
- Je vhodné minimalizovať dátový prenos medzi zariadením a hostiteľom
- V určitých prípadoch je vhodné niektoré dáta prepočítať na zariadení aj v prípade, že toto neprinesie výkonnostný zisk (ušetrí sa na prenose medzi hostiteľom a zariadením)
- Ideálny prístup do globálnej pamäte je splývajúci
- Minimalizácia prístupu do globálnej pamäte (dáta je vhodné prekopírovať do lokálnej)
- Je vhodné vyhnúť sa automatickej konverzii typu `float` na `double` (použiť `1.0f` namiesto `1.0`)
- Operácie `divide` a `modulo` nad celými číslami sú drahé (vyžadujú relatívne veľké množstvo procesorového času), pokiaľ možno, je vhodné nahradiť ich bitovými operáciami
- Pri vetvení (napríklad príkaz `if`) dochádza na GPU k vyhodnoteniu všetkých možných ciest, čo prináša spomalenie. V prípade, že sa v rámci jedného bloku vyskytnú rôzne cesty, dochádza k sériovému vykonaniu. Preto je nutné zabezpečiť, aby všetky vlákna v jednom bloku vykonávali tú istú časť podmienky.

7.6 Jednoduchá aplikácia

V rámci prvotného testu budú pomocou grafickej karty sčítané prvky dvoch vektorov (Tabuľka 3: Jednoduchý kernel). Kernel je umiestnený v samostatnom súbore `sum.cl`.

Tabuľka 3: Jednoduchý kernel

```
1:   __kernel void Sum(__global int *a, __global int *b, __global int *c)
2:   {
3:       uint id = get_global_id(0);
4:       for(int i=0;i<10;i++)
5:           c[i]=a[i]+b[i];
6:   }
```

Teraz je potrebné napísať obslužný program bežiaci na CPU. Ako prvé je nutné zahrnúť všetky hlavičkové súbory (v tomto prípade to sú `stdio.h` a `CL/cl.h`) a zadefinovať premenné, ktoré sa budú používať (Tabuľka 4: Úvodná inicializácia programu).

Tabuľka 4: Úvodná inicializácia programu

```
1:   #include <stdio.h>
2:   #include <CL/cl.h>
3:
4:   cl_context context = NULL;
5:   cl_command_queue commandQueue = NULL;
6:   cl_program program = NULL;
7:   cl_device_id device = NULL;
8:   cl_kernel kernel = NULL;
9:   cl_mem memObjects[3] = { 0, 0, 0};
10:  cl_int errNum;
```

Po týchto úvodných deklaráciách je potrebné zvoliť platformu, ktorá bude použitá (Tabuľka 5: Voľba platformy). Načíta sa zoznam všetkých dostupných platforiem (riadok 18), následne bude zvolená prvá z nich (riadok 21) a na nej vytvorený kontext pre všetky dostupné GPU (riadok 24).

Tabuľka 5: Voľba platformy

```
12:  cl_context CreateContext(){
13:        cl_int errNum;
14:        cl_uint numPlatforms;
15:        cl_platform_id firstPlatformId;
16:        cl_context context = NULL;
17:
18:        errNum = clGetPlatformIDs(1, &firstPlatformId, &numPlatforms);
19:            if (errNum != CL_SUCCESS || numPlatforms <= 0) return errNum;
20:
21:        cl_context_properties contextProperties[]=
22:        {CL_CONTEXT_PLATFORM,(cl_context_properties)firstPlatformId,0};
23:
24:        context = clCreateContextFromType(contextProperties, CL_DEVICE_TYPE_GPU,
25:         NULL, NULL, &errNum);
26:            if (errNum != CL_SUCCESS) return errNum;
27:        return context;
28:   }
```

V nasledujúcom kroku je potrebné vytvoriť na zariadení frontu príkazov pre daný kontext (Tabuľka 6: Inicializácia fronty príkazov). Najskôr sa zistí veľkosť zásobníka použitého zariadenia (riadok 37). Následne, ak je nenulový, je možné preň alokovať miesto pomocou príkazu `clGetContextInfo` (riadky 44 a 45). Potom sa pomocou príkazu `clCreateCommandQueue` (riadok 51) vytvorí požadovaná fronta príkazov. V závere sa uvoľní použitá pamäť (riadok 58).

Tabuľka 6: Inicializácia fronty príkazov

```
30:   cl_command_queue CreateCommandQueue(cl_context context,
31:    cl_device_id *device, int * maxCU){
32:       cl_int errNum;
33:       cl_device_id *devices;
34:       cl_command_queue commandQueue = NULL;
35:       size_t deviceBufferSize = -1;
36:
37:       errNum = clGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
38:       NULL, &deviceBufferSize);
39:       if (errNum != CL_SUCCESS) return NULL;
40:
41:       if (deviceBufferSize <= 0) return NULL;
42:
43:       devices = new cl_device_id[deviceBufferSize / sizeof(cl_device_id)];
44:       errNum = clGetContextInfo(context, CL_CONTEXT_DEVICES,
45:       deviceBufferSize, devices, NULL);
46:       if (errNum != CL_SUCCESS){
47:           delete [] devices;
48:           return NULL;
49:       }
50:
51:       commandQueue = clCreateCommandQueue(context, devices[0], 0, NULL);
52:       if (commandQueue == NULL){
53:           delete [] devices;
54:           return NULL;
55:       }
56:
57:       *device = devices[0];
58:       delete [] devices;
59:       return commandQueue;
60:   }
```

V ďalšom kroku sa skompiluje samotný program pre použité zariadenie (Tabuľka 7: Vytvorenie programu). Funkcia vyžaduje parameter s názvom súboru obsahujúcim program, ktorý sa načíta (riadok 69). Potom sa vytvorí program (riadok 73) a následne pomocou príkazu `clBuildProgram` sa skompiluje (riadok 77). V prípade zlyhania kompilácie dôjde k výpisu chýb do konzoly (riadok 82).

Tabuľka 7: Vytvorenie programu

```
62:   cl_program CreateProgram(cl_context context, cl_device_id device,
63:    const char* fileName){
64:       cl_int errNum; cl_program program;
65:
66:       std::ifstream kernelFile(fileName, std::ios::in);
67:       if (!kernelFile.is_open()) return NULL;
68:       std::ostringstream oss;
69:       oss << kernelFile.rdbuf();
70:
71:       std::string srcStdStr = oss.str();
72:       const char *srcStr = srcStdStr.c_str();
73:       program = clCreateProgramWithSource(context, 1,
74:       (const char**)&srcStr, NULL, NULL);
75:       if (program == NULL) return NULL;
76:
77:       errNum = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
78:       if (errNum != CL_SUCCESS){
79:           char buildLog[16384];
80:           clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
81:            sizeof(buildLog), buildLog, NULL);
82:           std::cerr << "Error in kernel: " << std::endl << buildLog;
83:           clReleaseProgram(program);
84:           return NULL;
85:       }
86:       return program;
87:   }
```

Taktiež je potrebná funkciu na uvoľnenie objektov, ktoré boli vytvorené (Tabuľka 8: Uvoľnenie použitej pamäti).

Tabuľka 8: Uvoľnenie použitej pamäti

```
89:   void Cleanup(cl_context context, cl_command_queue commandQueue,
90:   cl_program program, cl_kernel kernel, cl_mem memObjects[3]){
91:       for (int i = 0; i < 3; i++)
92:            if (memObjects[i] != 0) clReleaseMemObject(memObjects[i]);
93:       if (commandQueue != 0) clReleaseCommandQueue(commandQueue);
94:       if (kernel != 0) clReleaseKernel(kernel);
95:       if (program != 0) clReleaseProgram(program);
96:       if (context != 0) clReleaseContext(context);
97:   }
```

Ďalej je potrebné alokovať pamäť pre vstupné a výstupné premenné pre kernel (Tabuľka 9: Alokácia pamäte pre kernel) pomocou funkcie `CreateMemObjects()`. Ako je možné vidieť, v tomto prípade sa alokujú tri pamäťové priestory veľkosti desiatich prvkov typu `int`.

Tabuľka 9: Alokácia pamäte pre kernel

```
100:   bool CreateMemObjects(cl_context context, cl_mem memObjects[3],
101:    int *a, int *b, int *c){
102:        memObjects[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
103:         CL_MEM_COPY_HOST_PTR, sizeof(int)*10, a, NULL);
104:        memObjects[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
105:         CL_MEM_COPY_HOST_PTR, sizeof(int)*10, b, NULL);
106:        memObjects[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY |
107:         CL_MEM_COPY_HOST_PTR, sizeof(int)*10, c, NULL);
108:
109:        if (memObjects[0] == NULL || memObjects[1] == NULL
110:            || memObjects[2]== NULL) return false;
111:        return true;
112:   }
```

Taktiež je nutné nastaviť všetky argumenty pre kernel a overiť, že nastavenie prebehlo úspešne (Tabuľka 10: Nastavenie parametrov pre kernel).

Tabuľka 10: Nastavenie parametrov pre kernel

```
114:   errNum  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &memObjects[0]);
115:   errNum |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &memObjects[1]);
116:   errNum |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &memObjects[2]);
117:   errNum |= clSetKernelArg(kernel, 3, sizeof(cl_mem), &c);
118:   if (errNum != CL_SUCCESS) return -1;
```

Ako posledný krok sa zavolá kernel s počtom požadovaných vláken (Tabuľka 11: Volanie programu). Premenná `maxLocal` určuje počet súbežných vlákien pre jednu pracovnú skupinu, premenná `maxGlobal`, ktorá musí byť celým násobkom premennej `maxLocal`, vyjadruje celkový počet vykonaní nášho kernelu. Ten je volaný funkciou `clEnqueueNDRangeKernel()`, ktorej treba explicitne uviesť počet použitých dimenzií (riadok 126). Po ukončení výpočtu sa príkazom `clEnqueueReadBuffer` (riadok 128) získajú do premennej `result` výsledky. Jedným z parametrov tejto funkcie je aj veľkosť, ktorú z pamäti zariadenia budeme kopírovať (v tomto prípade sa jedná o 10x veľkosť typu `int`, keďže sa sčítavali polia o 10 prvkoch). Na záver je potrebné uvoľniť použité prostriedky (riadok 130) [13].

Tabuľka 11: Volanie programu

```
120:   context = CreateContext();
121:   commandQueue = CreateCommandQueue(context, &device, &maxCU);
122:   program = CreateProgram(context, device, "sum.cl");
123:   kernel = clCreateKernel(program, "Sum", NULL);
124:   size_t maxLocal[1]={10};
125:   size_t maxGlobal[1]={10};
126:   clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL,
127:   maxGlobal, maxLocal, 0, NULL, NULL);
128:   clEnqueueReadBuffer(commandQueue, memObjects[2],
129:    CL_TRUE, 0, 10*sizeof(int), result, 0, NULL, NULL);
130:   Cleanup(context, commandQueue, program, kernel, memObjects);
```

7.7 Zhrnutie kapitoly

V predchádzajúcich podkapitolách boli uvedené základné znalosti potrebné pre programovanie grafických kariet pomocou OpenCL. Taktiež boli rozobrané vhodné spôsoby prístupu k pamäti a spôsoby, ktorým je vhodné sa vyvarovať (konflikty prístupu do pamäte, zbytočná serializácia programu). Z uvedeného vyplýva, že pre programovanie grafických kariet za cieľom dosiahnutia maximálneho výkonu je nutné porozumieť ich architektúre, ktorá je výrazne odlišná od bežných procesorov. Taktiež bol ukázaný a vysvetlený základný kód potrebný na prístup k OpenCL zariadeniu a jeho programovaniu. Z uvedených zdrojových kódov je vidieť, že tento režijný kód má relatívne veľký rozsah (prakticky sa jedná o 130 riadkov zdrojového kódu, ktorý bude s ďalšími požiadavkami rásť), pričom boli uvedené iba tie najpotrebnejšie veci a kontrola správnosti argumentov bola vynechaná. Pri reálnom nasadení aplikácie by bolo nutné tento obslužný kód ďalej rozšíriť. Výhodou však je, že tento úvod je z väčšej časti rovnaký (k zmene dochádza iba pri alokácii pamäťových priestorov) pre všetky OpenCL zariadenia. Je preto možné jeho jednoduché a opakované použitie.

8 DEMONŠTRÁCIA VÝKONNOSTI GPU

Za účelom dokázania vhodnosti grafických kariet na SIMD výpočty bol zvolený algoritmus na dešifrovanie hesiel z MD5 šifry pomocou hrubej sily (jedná sa o šifrovanie určitého reťazca a následné porovnanie jeho výslednej šifry s hľadanou). Na základe verejne známeho princípu tohto šifrovacieho algoritmu [14] bola vytvorená jeho modifikovaná verzia, ktorá sa testuje na CPU aj GPU a zároveň sa meria čas potrebný na prelomenie jednoduchého hesla. Z neho sa určí počet spracovaných hesiel za sekundu a v závere porovná zrýchlenie programu pri behu na grafickej karte. Správnosť tejto implementácie bola potvrdená porovnaním s viacerými generátormi na webe, kde bola pre rôzne reťazce vygenerovaná kontrolná suma, ktorá bola následne prelomená pomocou tohto algoritmu. Ten pre všetky šifry našiel pôvodný reťazec.

8.1 Princíp algoritmu

Algoritmus má ako vstupné parametre hľadaný hash, reťazec, od ktorého sa začína hľadať a jeho dĺžku. Následne sa pre reťazec vypočíta hash a porovná sa s požadovaným. V prípade zhody sa do výsledku uloží poradové číslo nájdeného reťazca a hľadanie sa ukončí. Inak sa pokračuje v hľadaní s inkrementovanou hodnotou posledného znaku o jeden. Po nájdení zhody sa zrekonštruuje heslo a vypíše.

8.2 GPU verzia

Program automaticky vyberie prvú platformu v PC a nájde v nej dostupné grafické karty s podporou OpenCL. Vybraná bude prvá z nich, následne budú získané údaje o počte výpočtových jednotiek a maximálnom počte vlákien na každú z nich. Maximálny počet súbežných vlákien na výpočtovú jednotku je obmedzený hardwarom, avšak v prípade, že je kernel zložitejší, môže byť toto číslo menšie (z dôvodu obmedzených prostriedkov dostupných pre každú výpočtovú jednotku). Preto sa maximálny počet pracovných jednotiek nastaví až pri kompilácii kernela. Taktiež počet výpočtových jednotiek je daný použitým hardwarom, avšak v prípade použitia väčšej dôjde k opakovanému použitiu jednotiek, až kým nebudú vykonané všetky požiadavky.

Aplikácia automaticky nastavuje túto maximálnu hodnotu ako počet pracovných jednotiek a hodnotu pracovných skupín ako počet výpočtových jednotiek. Pre každé vlákno dochádza k výpočtu 26^4 šifier (skúšajú sa všetky možné kombinácie malých písmen abecedy na posledných 4 pozíciách v danom reťazci) od reťazca vypočítaného na základe jedinečného ID pre každý kernel. Celkovo prebehne v jednom cykle na grafickej karte 26^4 * pracovné jednotky * pracovné skupiny výpočtov. Z dôvodu problematického ukončovania bežiacich vlákien bolo zvolené riešenie, kedy sa v prípade nájdenia hesla do výsledku zapíše jeho poradové číslo v danom vlákne a počíta sa ďalej. Keďže hlavným parametrom na

porovnanie je počet spracovaných hesiel za sekundu, ničomu to neprekáža. Po prebehnutí výpočtu sa na CPU načíta výsledok, ak heslo nebolo nájdené, celý proces sa opakuje od ďalšieho štartovacieho reťazca.

Program sa nepodarilo optimalizovať do maximálnej možnej miery, keďže nebolo možné vyhnúť sa serializácii bloku v prípade nájdenia výsledku. Tá vznikla ako dôsledok vetvenia v príkaze `if`, ktorý sa vykoná iba vo vlákne, ktoré našlo heslo, ostatné vlákna však tento príkaz nevykonávajú.

Kompilátor má nastavené nasledujúce optimalizácie:

- `-cl-single-precision-constant` - všetky konštanty majú jednoduchú presnosť
- `-cl-denorms-are-zero` - zaokrúhlenie veľmi malých čísel na nulu (exponent je na hranici rozlíšenia)
- `-cl-strict-aliasing` - ukazovatele rozdielnych typov neukazujú na rovnaké miesto v pamäti
- `-cl-mad-enable` - umožňuje výpočet `a * b + c` pomocou mad jednotky, ktorá môže mať zníženú presnosť
- `-cl-no-signed-zeros` - nerozlišuje sa medzi `-0.0` a `+0.0`
- `-cl-fast-relaxed-math`- predpokladá, že v priebehu výpočtu sa nevyskytne výsledok s hodnotou nekonečno, alebo hodnotou, ktorá nie je číslo; toto môže porušovať štandard IEEE 754

8.3 CPU verzia

V princípe funguje rovnako ako GPU verzia, kedy dochádza k volaniu cyklu o veľkosti `pracovné jednotky * pracovné skupiny`, ktorý obsahuje ďalší cyklus o veľkosti 26^4 opakovaní. Po skončení dochádza k vyhodnoteniu, či bola nájdená zhoda šifier. V prípade neúspechu sa celý výpočet opakuje, tentokrát s nasledujúcim začiatočným reťazcom.

Na využitie maximálneho výkonu CPU som využil knižnicu OpenMP, pomocou ktorej sa prvý cyklus algoritmu vykonáva paralelne vo vláknach podľa počtu jadier procesora. Taktiež boli vo Visual Studiu 2012 zapnuté nasledujúce optimalizácie kódu:

- `/Ox` - plná optimalizácia
- `/Ot` - uprednostnenie rýchlosti kódu pred veľkosťou
- `/GL` - celková optimalizácia
- `/fp:fast` - operácie s plávajúcou desatinou čiarkou – uprednostniť rýchlosť

Zapnutie využitia inštrukčnej sady SSE a SSE2 prinieslo pokles výkonu, preto nebolo vo finálnej verzii použité.

8.4 Namerané výsledky

Aplikáciu bola otestovaná na hardware z Tabuľka 12: Výkonnostné porovnanie CPU a GPU pre MD5 algoritmus.

Tabuľka 12: Výkonnostné porovnanie CPU a GPU pre MD5 algoritmus

		Cena v čase nákupu[€] [15]	Počet hesiel [mil/s]	Teoretický výkon [GFLOPS]
CPU	Intel Core 2 Quad Q6600 @ 3,21 GHz	250 (2/2008)	25,60	47
	Intel Pentium E5400 @ 2,7 GHz	60 (2/2010)	11,48	20
	Intel Core i5 2500K @ 3,6 GHz	200 (7/2011)	23,11	46
	Intel Core i5 430M @ 2,26 GHz	- (8/2010)	17,31	15
	Intel Core 2 Duo E8500 @ 3,16 GHz	170 (3/2008)	13,93	23
GPU	NVIDIA GTX 560Ti 448 (GF110) @ 932/1864/4514 MHz	280 (2/2012)	1404,69	1500
	ATI Radeon 5750 @ 800/4000 MHz	120 (2/2010)	272,69	1100
	NVIDIA GTX 560Ti (GF114) @ 981/1962/4050/MHz	200 (7/2011)	950,54	1400
	ATI Mobility Radeon HD 5650 @ 550/1600 MHz	- (8/2010)	90,14	520
	NVIDIA 9600GT @ 650/1625/1800 MHz	160 (3/2008)	160,35	210

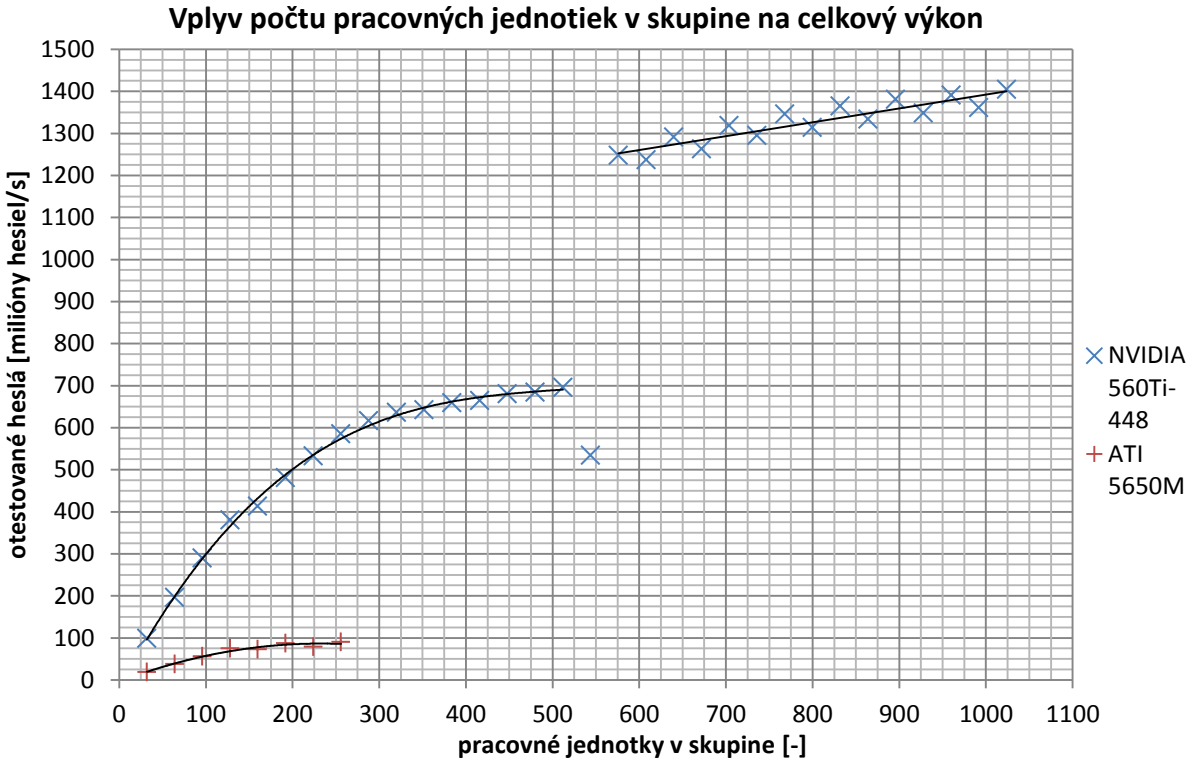
Z Tabuľka 12: Výkonnostné porovnanie CPU a GPU pre MD5 algoritmus je vidieť, že aj najpomalšia z testovacích grafický kariet prekonáva najrýchlejší z testovacích procesorov viac ako trojnásobne, čo sa dá považovať za jednoznačnú vhodnosť GPU na takýto typ výpočtov.

Ďalej je vidieť, že pre architektúru grafických kariet ATI daný algoritmus nie je vhodný (dôvodom je obmedzenie počtu pracovných jednotiek na skupinu na hodnotu 256, avšak u kariet NVIDA je to až 1024) a oproti kartám NVIDIA je pomer počet šifier za sekundu/teoretický výkon výrazne menší ako pomer ich teoretických výkonov. Dá sa predpokladať, že s ďalšou optimalizáciou pre kernel by bolo možné u grafických kariet ATI

dosiahnuť výkonnosť kariet NVIDIA. Taktiež voľba iného prístupu k riešeniu problému by mohla priniesť lepšie výsledky na oboch architektúrach.

Pri porovnaní procesorov sa ukazuje, že algoritmus nevyužíva inštrukcie pridávané do najnovších procesorov, pretože napriek rozdielu 3,5 roka medzi architektúrami Intel Core 2 Quad Q6600 a Intel Core i5 2500K, je novší paradoxne pomalší. Tento jav je spôsobený vysokým pretaktovaním prvého menovaného, čím došlo k razantnému zvýšeniu priepustnosti vnútorných L1 a L2 cache pamätí až nad úroveň menej pretaktovaného novšieho procesora.

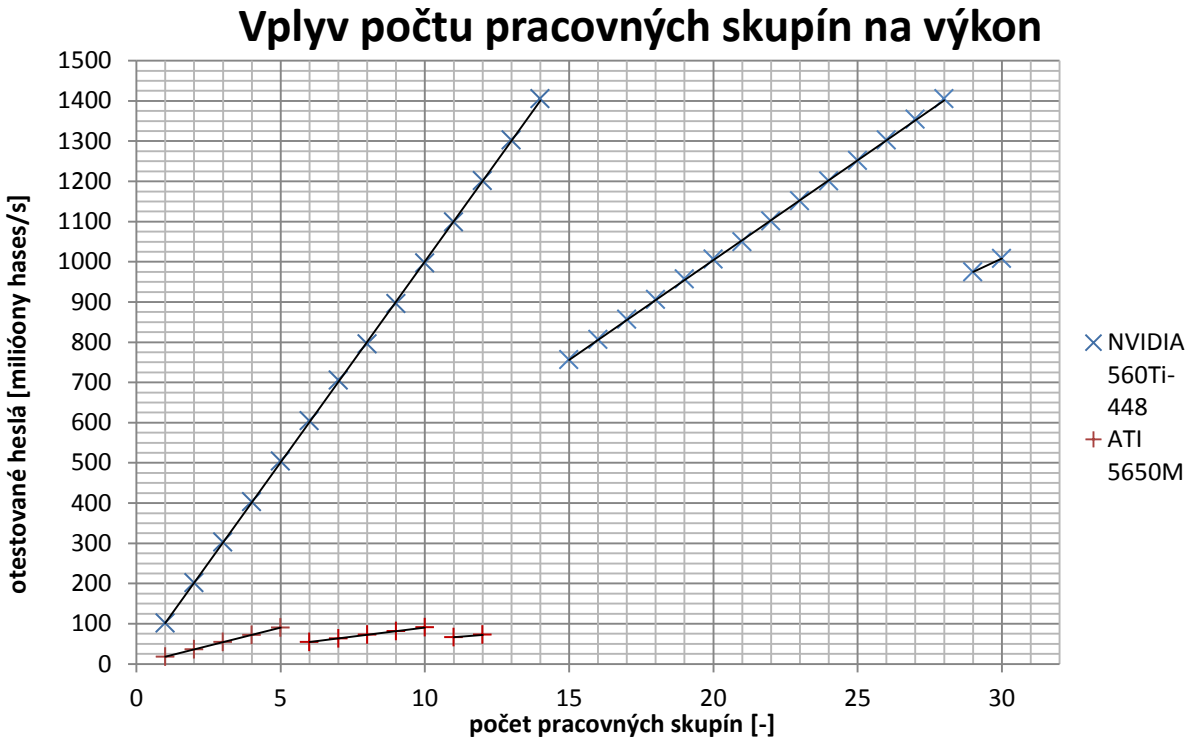
Na platforme Windows dochádzalo v niektorých prípadoch k automatickému obnoveniu ovládačov grafickej karty, ktorý považoval dočasné nereagovanie systému spôsobené náročným výpočtom (nástroj MSI Afterburner vypočítal vyťaženie GPU na viac ako 99%) na grafickej karte za chybu. Z tohto dôvodu bolo nutné v registroch systému toto automatické obnovenie zakázať (zmena, prípadne vytvorenie kľúča `TdrLevel` typu `REG_DWORD` na hodnotu `0` na adrese `HKLM\System\CurrentControlSet\Control\GraphicsDrivers`). Táto zmena je však firmou Microsoft odporúčaná iba na testovacie účely.



Obrázok 11: Vplyv počtu pracovných jednotiek v skupine na celkový výkon

Z grafu na Obrázok 11: Vplyv počtu pracovných jednotiek v skupine na celkový výkon je vidieť, že vplyv oneskorenia prístupu ku globálnej pamäti je výrazný, keďže so zvyšujúcim sa počtom pracovných jednotiek v skupine dochádza k výraznému nárastu výkonu. Výrazný prekmit a následný skok u grafickej karty NVIDIA 560TI-448 je spôsobený dosiahnutím limitu súbežne bežiacich jednotiek (vlákien) na výpočtovú jednotku. Ďalšie zvýšenie

spôsobilo väčší rozptyl prístupu k pamäti, čo umožnilo rast výkonu. Je tiež vidieť, že pri približovaní sa maximálnemu počtu povolených pracovných jednotiek dochádza k prekmitom. U karty NVIDIA za to pravdepodobne môže problematický prístup ku globálnej pamäti, u karty ATI je to spôsobené nedodržaním odporúčania výrobcu, že počet pracovných jednotiek v skupine má byť násobkom čísla 64.



Obrázok 12: Vplyv počtu pracovných skupín na výkon

Z grafu na Obrázok 12: Vplyv počtu pracovných skupín na výkon je vidieť, že NVIDIA 560Ti-448 má 14 výpočtových jednotiek a ATI 5650M 5. Taktiež sa dá usudzovať, že návrh algoritmu je dobrý, nakoľko s vyžívaním ďalších výpočtových jednotiek výkon škáluje lineárne u oboch kariet (každej výpočtovej jednotke sa priradí jedna pracovná skupina). Po prekročení počtu výpočtových jednotiek počtom pracovných skupín dochádza k prepadu výkonu, čo je dané faktom, že ďalšie pracovné skupiny sa nedajú rozdeliť medzi voľné výpočtové jednotky a časť výkonu GPU sa preto nedá využiť.

Na základe grafov z Obrázok 11: Vplyv počtu pracovných jednotiek v skupine na celkový výkon a Obrázok 12: Vplyv počtu pracovných skupín na výkon je jasné, že najvhodnejší počet pracovných skupín je celý násobok počtu výpočtových jednotiek karty a počet pracovných jednotiek v skupine je maximálna hodnota, ktorú povolí kompilátor (uvedené súvislosti nemusia nevyhnutne platiť pre všetky programy, je nutné testovať).

9 KOMPLEXNEJŠIA ÚLOHA

Ako komplexnejšia úloha bola realizovaná neurónová sieť na rozpoznávanie ručne písaných čísel, a ich následný prevod do elektronickej podoby (tzv. OCR). Vzhľadom na to, že predmetom tejto práce nie je implementácia samotnej neurónovej siete, ale výkonnostné porovnanie procesorov architektúry x86 a grafických kariet, bola použitá neurónová sieť z online zdroja [16], ktorú bola následne upravená tak, aby ju bolo možné ľahko implementovať na GPU. Z tohto dôvodu taktiež nebudú v tejto práci podrobne popisované princípy neurónových sietí, ale budú iba načrtnuté. Podrobnejšie informácie, vrátane matematických definícií je možné nájsť v [17] a [18].

9.1 Stručný princíp neurónovej siete

Tento popis nie je platný pre všetky typy neurónových sietí a zaoberá sa iba princípom feed-forward (šírenie vpred), kde každá vrstva neurónov ovplyvňuje iba nasledujúcu vrstvu.

Základným princípom neurónovej siete je, že pre určité vstupy produkuje na základe váhového vektoru konkrétne výstupy. Tieto vyjadrujú pravdepodobnosť, s akou je daný vstup konkrétnym výstupom. Takáto sieť nemusí byť nutne dvojvrstvová (vstup – výstup), ale môže obsahovať ďalšie skryté vrstvy, kde pre každý pár týchto vrstiev existuje samostatný váhový vektor. Pravdepodobnosť na výstupe je vyjadrená ako číslo v definovanom rozsahu, najčastejšie sa používa rozsah 0 – 1 a -1 – 1. Ideálnym stavom je, aby pre konkrétne vstupy neboli pravdepodobnosti roztrúsené v celom rozsahu, ale aby maximálna pravdepodobnosť bola práve pri požadovanom výstupe a minimálna pravdepodobnosť pri všetkých ostatných.

Toto správanie sa dosahuje takzvaným učením neurónovej siete, kedy sa upravuje vektor váh, ktorý sa na počiatku inicializuje náhodnými hodnotami v určitom rozsahu. Z nich sa na základe vstupov počítajú výstupy. Spôsobov učenia existuje niekoľko, v tomto prípade bol implementovaný algoritmus spätnej propagácie (backpropagation). Tento algoritmus funguje na princípe spätného šírenia chyby na základe odchýlok výstupného vektora od vektora požadovaných hodnôt. Tieto chyby prakticky vyjadrujú, akou časťou sa konkrétny neurón podieľa na danom výstupe. V závere sa na základe týchto chybových vektorov upravia všetky váhové vektory, kedy sa k ich pôvodnej hodnote pričíta pôvodný vstupný vektor násobený práve týmto chybovým vektorom. Následne sa môže cyklus opakovať.

Prepojenie medzi vrstvami siete môže byť úplné, kedy každý neurón z jednej vrstvy je spojený váhou s každým neurónom vrstvy nasledujúcej, alebo neúplné, kedy sú tieto spojenia definované na základe určitého princípu.

9.1.1 Pojmy používané v neurónových sietiach

Koeficient učenia (learning rate) - Číslo, ktorým sa násobí korekcia váh. Používa sa, aby nedošlo k príliš veľkej zmene váhy a následnému zväčšeniu chyby. Je menší ako jedna, jeho konkrétna hodnota sa často určuje experimentálne na základe zložitosti siete

Aktivačná funkcia – Funkcia, ktorá normalizuje vstupnú hodnotu na výstup v požadovanom rozsahu. Najčastejšie sa používa sigmoid a hyperbolický tangens z dôvodu relatívne jednoduchého výpočtu ich derivácie, ktorá je potrebná pri spätnom šírení chyby.

Bias – Špecialna váha vo váhovom vektore, ktorá nie je napojená na vstupný neurón. Má vždy hodnotu jedna a jeho váha sa nikdy nemení. Slúži na stabilizáciu výstupu.

Tréningová množina – Sada unikátnych vstupov a k nim príslušných výstupov, pomocou ktorých učíme neurónovú sieť.

Epocha – cyklus, kedy boli neurónovej sieti na učenie predložené všetky vzory z tréningovej množiny

Online learning – najčastejšie používaný spôsob učenia pri použití backpropagation, kedy sa váhové vektory upravujú okamžite po spočítaní chyby.

Batch learning – k úprave váh dochádza až po dokončení celej epochy. Vo všeobecnosti sa tento spôsob používa pri paralelizácii učenia. Nevýhodou je pomalšia konvergencia ako pri online učení.

Chyba siete – vážený priemer chýb všetkých výstupných neurónov pre celú tréningovú množinu

Natrénovanie siete (nájdenie/nastavenie správnych hodnôt vektorov váh) trvá typicky 1000 a viac epoch. Učenie sa typicky končí po dosiahnutí menšej ako požadovanej chyby pri predkladaní tréningovej množiny.

Podrobnejšie informácie, vrátane matematických definícií je možné nájsť na [17] a [18].

9.2 Princíp algoritmu

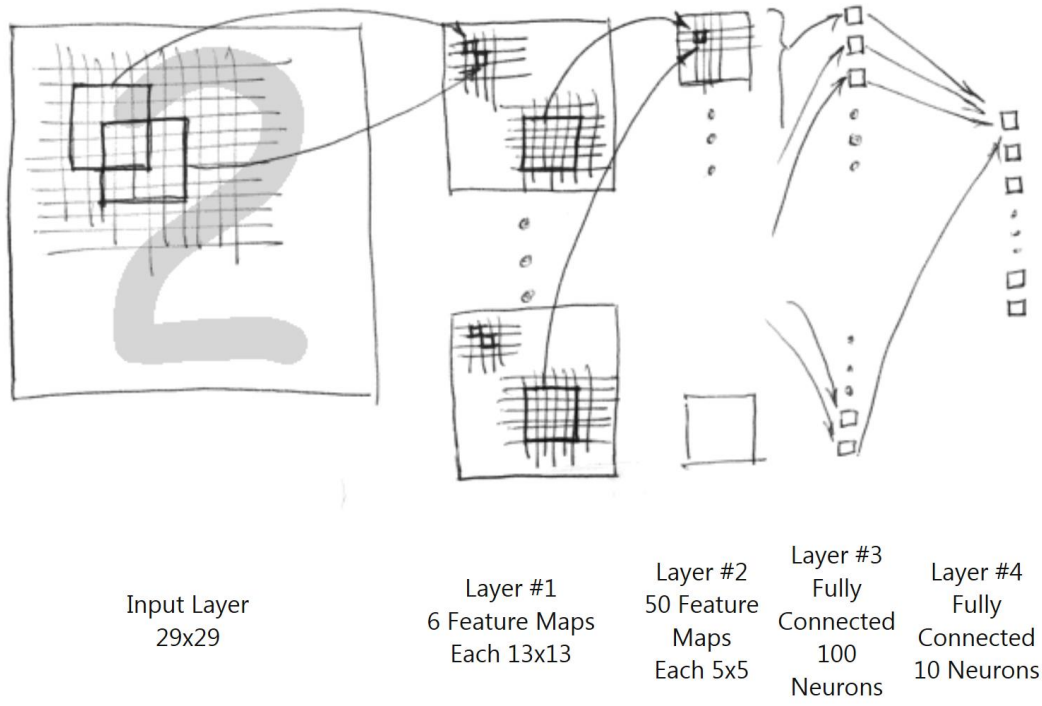
Implementovaná neurónová sieť je päťvrstvová a konvolučná (neurónová sieť obsahuje viac spojení ako váh, čo zrýchľuje učenie - dochádza k zníženiu výpočtovej a pamäťovej náročnosti použitím rovnakej váhy pre rôzne spojenia). Každý pár vrstiev je riešený ako jedna funkcia, preto výpočtovú časť siete tvoria štyri funkcie. Ako vstup slúži binárny obraz o rozmere 29x29 pixelov obsahujúci ručne písané číslo, kde hodnota 0 reprezentuje čiernu farbu (samotný znak) a 1 bielu farbu (okolie znaku). Toto sú zároveň hodnoty vstupných neurónov, ktorých je 841. Všetky neuróny a váhy sú reprezentované ako vektory. Cieľom je, aby pre výstupy výstupnej vrstvy mali hodnotu 0,8 pre neurón prislúchajúci požadovanému výsledku, a hodnotu -0,8 pre všetky ostatné neuróny.

Prvá skrytá vrstva je konvolučná vrstva obsahujúca 6 rôznych máp o veľkosti 13x13 pixelov (neurónov), kde každý neurón je počítaný z okna o veľkosti 5x5 pixelov zo vstupnej vrstvy, pričom je každý druhý pixel preskočený. Z toho vyplýva, že druhá vrstva neurónov (prvá skrytá) má 13 * 13 * 6 = 1014 neurónov a (5 * 5 + 1) * 6 = 156 váh. Keďže však každý neurón má 26 spojení, celkový počet spojení medzi vrstvami je 1014 * 26 = 26364. Toto je jeden z benefitov konvolučnej siete, keďže počet váh ktoré je potrebné trénovať je výrazne menší ako počet spojení, čím sa výrazne skracuje doba potrebná k tréningu.

Druhá skrytá vrstva je taktiež konvolučná, avšak obsahuje už 50 máp o rozmere 5x5. Každý neurón je počítaný ako okno z predchádzajúcej vrstvy o veľkosti 5x5 pixelov zo všetkých šiestich máp. Vo výsledku preto dostávame 5 * 5 * 50 = 1250 neurónov druhej skrytej vrstvy. Váhový vektor má (5 * 5 + 1) * 6 * 50 = 7800 unikátnych váh a celkov existuje s predchádzajúcou vrstvou 1250 * 26 = 32500 spojení.

Tretia skrytá vrstva je už plne prepojená a obsahuje 100 neurónov. Preto je každý neurón prepojený s 1250 neurónmi predchádzajúcej vrstvy. Z toho vyplýva 100 * (1250 + 1) = 125100 váh a zároveň aj počet spojení.

Posledná vrstva je vrstva výstupná o veľkosti 10 neurónov. Je taktiež plne prepojená, preto obsahuje 10 * (100 + 1) = 1010 váh a spojení. V prípade, že sieť nebude slúžiť iba na rozpoznávanie, ale aj na učenie, je tu ešte obsiahnutý výpočet chyby siete porovnaním s požadovanými hodnotami.



Obrázok 13: Vizualizácia klasifikácie neurónovej siete [16]

Týmto končí neurónová sieť a nasleduje ďalších sedem funkcií slúžiacich na učenie siete. Prvé tri slúžia na spätné šírenie chyby neurónov. Musia byť spolu s predchádzajúcimi štyrmi

výpočtovými počítané sekvenčne, nakoľko na sebe priamo závisia. Po nich nasledujú štyri funkcie, ktoré na základe vypočítaných chýb a koeficientu učenia upravujú jednotlivé dátové vektory. Tieto funkcie už môžu bežať paralelne (úlohový paralelizmus).

Celý tento cyklus bol paralelizovaný, čím dochádza k implementácii batch learningu. Z tohto dôvodu bola pridaná ešte jedna funkcia, ktorá zo všetkých výstupných vektorov vyberie iba hodnoty s najvyššou pravdepodobnosťou a naplní vektor konkrétnymi rozpoznanými hodnotami.

Celková paralelizácia tejto neurónovej siete je obmedzená svojou architektúrou, čo sa prejavuje pri učení pomocou GPU, kedy grafická karta nie je pri malých objemoch dát plne vyťažená.

9.3 Implementácia

Program vyžaduje ako vstupné parametre príkazového riadku prepínač požadovanej funkcie a súbor vstupných dát, voliteľne je možné zadať parameter určujúci počet znakov zo vstupného súboru, ktorý sa v sieti použije, toto číslo však musí byť násobkom 100.

Súbor obsahujúci vstupné dáta musí mať nasledujúci formát:

- Jeden znak je reprezentovaný ako binárna postupnosť 841 čísel (29x29 pixelov)
- Každá postupnosť reprezentujúca jeden znak sa musí opakovať 10-krát (daná postupnosť môže byť samozrejme odlišná)
- Počet rôznych znakov musí byť 10. Z toho vyplýva základná množina 100 postupností reprezentujúcich znaky
- Počet množín môže byť ľubovoľný
- Počet znakov v súbore sa rozpoznáva automaticky
- Súbor je v štandardnom textovom formáte (ASCII)
- Všetky dáta v súbore musia byť na jednom riadku

Tieto obmedzenia vstupného súboru sú z dôvodu automatického generovania vektoru požadovaných hodnôt použitých pri učení.

V prípade nezadania vstupných parametrov program spustí učenie siete na maximálne 100 prvých znakov. Prepínač `-c` slúži na samotnú klasifikáciu digitalizovaných čísel, kedy sa zistí veľkosť súboru, a z neho počet vstupných znakov. Prepínač `-l` slúži na učenie siete, kedy sa vykoná 1000 epoch. Na základe toho sa alokuje pamäť pre všetky potrebné vektory (polia) a následne sa naplní zo súboru vstupná vrstva a z ďalších súborov sa buď načítajú váhy neurónovej siete, alebo sa náhodne vygenerujú. Vektory pre chyby a vektory ostatných vrstiev nie je nutné inicializovať.

Nasleduje inicializácia grafickej karty, kedy sa skompiluje program, nastaví kontext zariadenia a prekopírujú sa dáta. Potom sa pokračuje volaním funkcie zabezpečujúcej výpočet na CPU a meria sa potrebný čas. Po skončení sa podobne volá GPU verzia algoritmu, kedy sa opäť meria čas. Po dobehnutí oboch verzií dochádza k uloženiu váh na disk, pre neskoršie použitie. V závere sa vypíšu výsledky a na základe času behu oboch verzií sa vypočíta násobok zrýchlenia výpočtu na grafickej karte.

Program [16] bol z dôvodu jednoduchej implementácie pre GPU prepísaný z C++ do čistého C, bez štruktúr a dynamického prideľovania pamäte. Z tohto dôvodu nie sú väzby medzi neurónmi držané v zásobníku, ale sú vypočítavané pomocou prístupovej matice a pre každú vrstvu musí na rozdiel od pôvodnej implementácie existovať osobitná funkcia.

9.4 GPU verzia

Implementácia na hostiteľskom zariadení je v princípe rovnaká ako pri predchádzajúcich programoch. Jediný rozdiel je využitie až jedenástich kernelov (pre každú vyššie popísanú funkciu jedna), z čoho vyplýva obšírnejšie alokovanie pamäte. Z dôvodu využitia úlohového paralelizmu pri kerneloch aktualizujúcich váhy bola použá pre každý kernel jedna fronta (dokopy štyri). V prípade učenia dochádza k opakovanému volaniu jednotlivých kernelov (okrem výstupného) v cykle a celá tréningová množina je spracovaná daným kernelom naraz. V každom cykle sa na konzolu vypisuje aktuálna chyba siete. Táto hodnota je však iba orientačná, nakoľko na zistenie presnej hodnoty by bolo nutné použitie atomických operácií. Pri ich použití dochádza k serializácii programu, čím sa výrazne znižuje výkon. Z tohto dôvodu bolo zvolené riešenie, pri ktorom vznikajú race conditions a spôsobuje na pohľad výraznú osciláciu celkovej chyby, aj keď tá v skutočnosti plynule klesá, podobne ako pri CPU verzii. Táto skutočnosť však neovplyvňuje samotné učenie a ostané výpočty v sieti. Hodnota celkovej chyby sa totižto pre žiadne ďalšie výpočty nepoužíva, nakoľko každé vlákno pracuje s vlastnou chybou, pri ktorej nevznikajú race conditions. Výpis orientačnej celkovej chyby je preto možné považovať za bonus a nie za chybu. Po prebehnutí všetkých epoch (1000) sa volá posledný kernel, ktorý spočíta výstupný vektor a dôjde k vypísaniu výsledkov do konzoly.

Program má opäť zapnuté všetky optimalizácie, ktoré sú v OpenCL možné. Ich zoznam a popis je v kapitole 8.2: GPU verzia. Každý kernel je optimalizovaný tak, aby prístup do pamäte grafickej karty spĺňal princíp popísaný v kapitole 7.3: Prístup do globálnej pamäte GPU, kedy všetky čítania pamäte pristupujú do jedného stĺpca, čím sa zabezpečuje maximálny výkon.

Prideľovanie práce pre GPU je opäť dynamické, kedy program zistí maximálny počet pracovných skupín pre použitú grafickú kartu a na jeho základe vypočíta najvhodnejšie rozdelenie práce za účelom dosiahnutia maximálneho výkonu.

9.5 CPU verzia

Verzia pre procesor je prakticky totožná s verziou pre GPU, avšak implementované optimalizácie prístupu do pamäti u CPU neprinášajú žiaden výkonnostný zisk. Funkcie majú pomocou cyklov `for` simulovanú blokovú organizáciu, ktorá je použitá na grafickej karte. Každá funkcia je paralelizovaná pomocou OpenMP, preto automaticky dochádza k využitiu všetkých dostupných procesorových jadier. Počet epoch je znovu 1000 a taktiež po každej dochádza k výpisu aktuálnej chyby siete. Táto je tentokrát už presná, nakoľko použitím atomickej operácie nedošlo k poklesu výkonu. Po prebehnutí cyklu sa znovu vypíšu výsledky do konzoly.

Program má opäť zapnuté optimalizácie dostupné v prostredí Microsoft Visual Studio 2012. Ich zoznam a popis je uvedený v kapitole 8.3: CPU verzia.

9.6 Komplikácie pri vývoji

Z dôvodu komplikovaného ladenia programu pre GPU bola ako prvá implementovaná CPU verziu programu. Vývoj však hneď od začiatku sprevádzali komplikácie, nakoľko algoritmus pre backpropagation bolo nutné vymyslieť prakticky od nuly, keďže nebolo možné používať zásobník na udržiavanie spojení. Reverzovanie výpočtu jednotlivých vrstiev bolo preto pomerne náročné a bolo ľahké spraviť chybu pri implementácii algoritmu, ktorá sa v závere prejavila ako neschopnosť siete učiť sa. Ďalším problémom bol výskyt javu zvaného catastrophic forgetting, kedy sa sieť síce učila, ale bola schopná sa naučiť rozpoznať iba jeden tréningový vzor. V prípade predloženia nového vzoru dochádzalo k „zabudnutiu“ vzoru predchádzajúceho. Tento problém sa podarilo vyriešiť vhodným nastavením koeficientu učenia a vhodnými rozsahom náhodných inicializačných váh.

Pri paralelizovaní funkcií bolo nutné niektoré z nich prepísať, nakoľko štandardne dochádzalo k problémom typu race conditions. Tieto sa pri štvorjadrovom procesore neprejavovali a sieť sa učila, problém však nastal pri implementácii pre GPU, kedy sa sieť neučila vôbec. Týmto však došlo k zníženiu miery paralelizácie pri posledných štyroch kerneloch, čo sa negatívne podpísalo na výkone.

Ďalší problém, ktorý sa prejavil iba pri GPU verzii, súvisel s použitím koeficientu učenia. V tomto prípade nebol dereferencovaný pointer, čím sa násobili hodnoty pre aktualizáciu váh adresou hodnoty namiesto hodnotou samotnou. Na tento fakt však kompilátor neupozornil a nájsť danú chybu zabralo veľa času.

Za závažný problém je možné považovať jav, kedy pri chybe v kerneli (najčastejšie prístup mimo alokovanú pamäť) dochádza k zníženiu frekvencií GPU a pamäte grafickej karty na hodnoty definované pre 2D mód. Týmto dochádza k výraznému poklesu výkonu. Nastaviť pôvodné 3D frekvencie je možné reštartovaním počítača alebo reštartovaním grafického

ovládača, čo je možné dosiahnuť zakázaním a opätovným povolením ovládača grafickej karty v správcovi zariadení.

9.7 Namerané výsledky

Výsledná aplikácia bola testovaná na tom istom hardware aký je uvedený v kapitole 8.4. Vzhľadom na zanedbateľné časy pri samotnom rozpoznávaní znakov (pri 1000 znakoch menej ako 1 sekunda pre CPU aj GPU), bolo testované iba učenie neurónovej siete, ktoré je zdĺhavým a náročným procesom. Toto je vidieť aj z Tabuľka 13: Výkonnostné porovnanie CPU a GPU pri učení neurónovej siete.

Tabuľka 13: Výkonnostné porovnanie CPU a GPU pri učení neurónovej siete

		Cena v čase nákupu[€] [15]	Počet hesiel [mil/s]	Teoretický výkon [GFLOPS]
CPU	Intel Core 2 Quad Q6600 @ 3,21 GHz	250 (2/2008)	994,21	47
	Intel Pentium E5400 @ 2,7 GHz	60 (2/2010)	3693,24	20
	Intel Core i5 2500K @ 3,6 GHz	200 (7/2011)	746,67	46
	Intel Core i5 430M @ 2,26 GHz	- (8/2010)	3842,07	15
	Intel Core 2 Duo E8500 @ 3,16 GHz	170 (3/2008)	38,04	23
GPU	NVIDIA GTX 560Ti 448 (GF110) @ 932/1864/4514 MHz	280 (2/2012)	296,17	1500
	ATI Radeon 5750 @ 800/4000 MHz	120 (2/2010)	57,26	1100
	NVIDIA GTX 560Ti (GF114) @ 981/1962/4050/MHz	200 (7/2011)	419,72	1400
	ATI Mobility Radeon HD 5650 @ 550/1600 MHz	- (8/2010)	994,21	520
	NVIDIA 9600GT @ 650/1625/1800 MHz	160 (3/2008)	3693,24	210

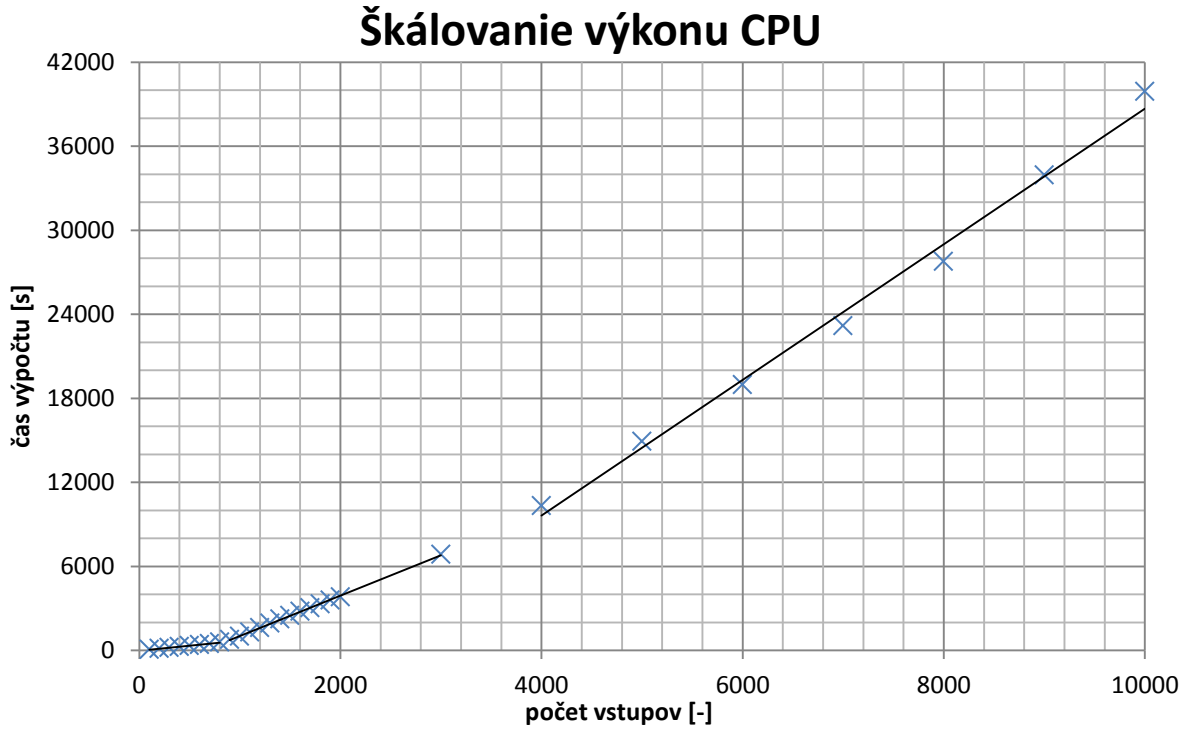
Z tabuľky je vidieť, že najslabšia testovaná grafická karta opäť prekonáva najvýkonnejší procesor, tentokrát však už ani nie dvojnásobne. Na prvý pohľad to síce nie je veľa, ale pri zložitých sieťach, s veľkou vstupnou množinou a tisíckami epoch, je aj dvojnásobné zrýchlenie žiadané. Tým sa ukazuje vhodnosť grafických kariet na učenie neurónovej siete.

Tabuľka taktiež ukazuje, že na rozdiel od predchádzajúceho programu, je táto implementácia neurónovej siete vhodná aj pre grafické karty AMD (ATI).

Porovnanie procesorov tentokrát ukazuje, že algoritmus už využíva inštrukcie pridávané do najnovších procesorov (na rozdiel od predchádzajúceho programu), kedy Intel Core i5 2500K podáva o 25% vyšší výkon ako pretaktovaný Intel Core 2 Quad Q6600.

9.8 Podrobné porovnanie škálovania výkonu CPU a GPU

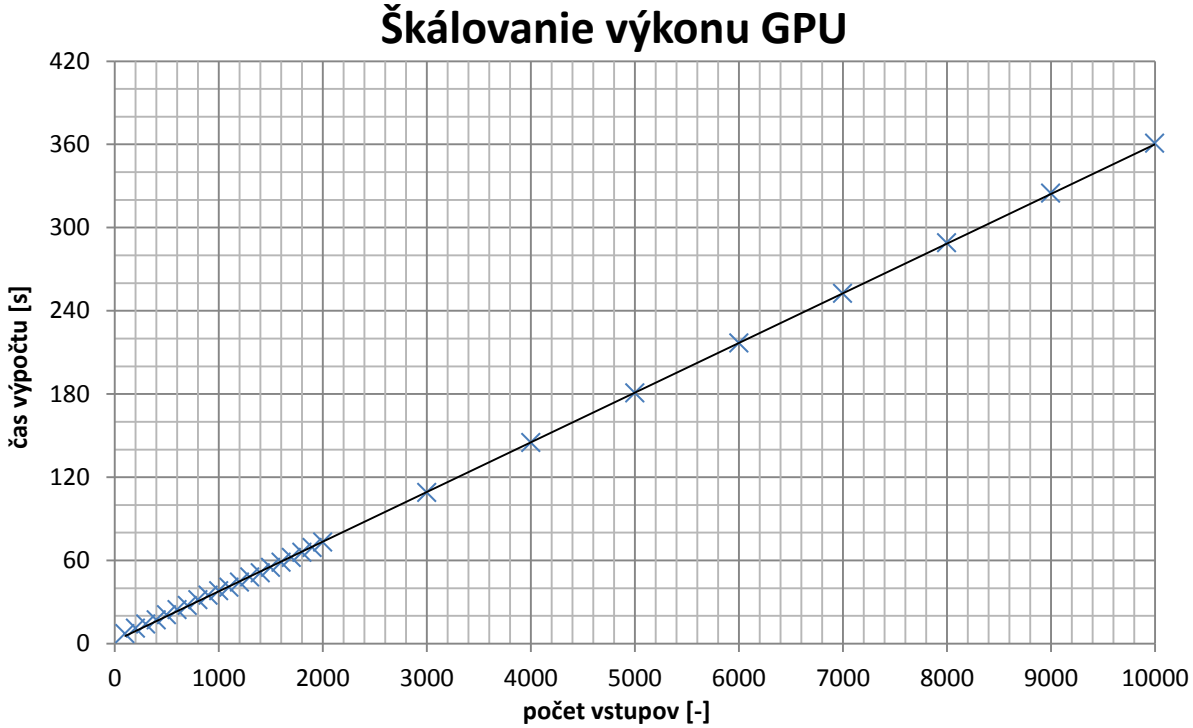
Vzhľadom na to, že pri učení pomocou grafickej karty NVIDIA GTX 560Ti – 448 nedochádzalo k plnému vyťaženiu GPU, bolo otestované škálovanie výkonu na rôznych počtoch vstupných dát. Toto škálovanie sa následne porovnalo so škálovaním procesoru Intel Core 2 Quad Q6600. Počet vstupov sa menil v rozsahu 100 až 10000, pričom po 2000 vzoriek bol krok 100. Následne, z dôvodu časovej náročnosti, bol zväčšený na 1000.



Obrázok 14: Škálovanie výkonu CPU pri 1000 epochách

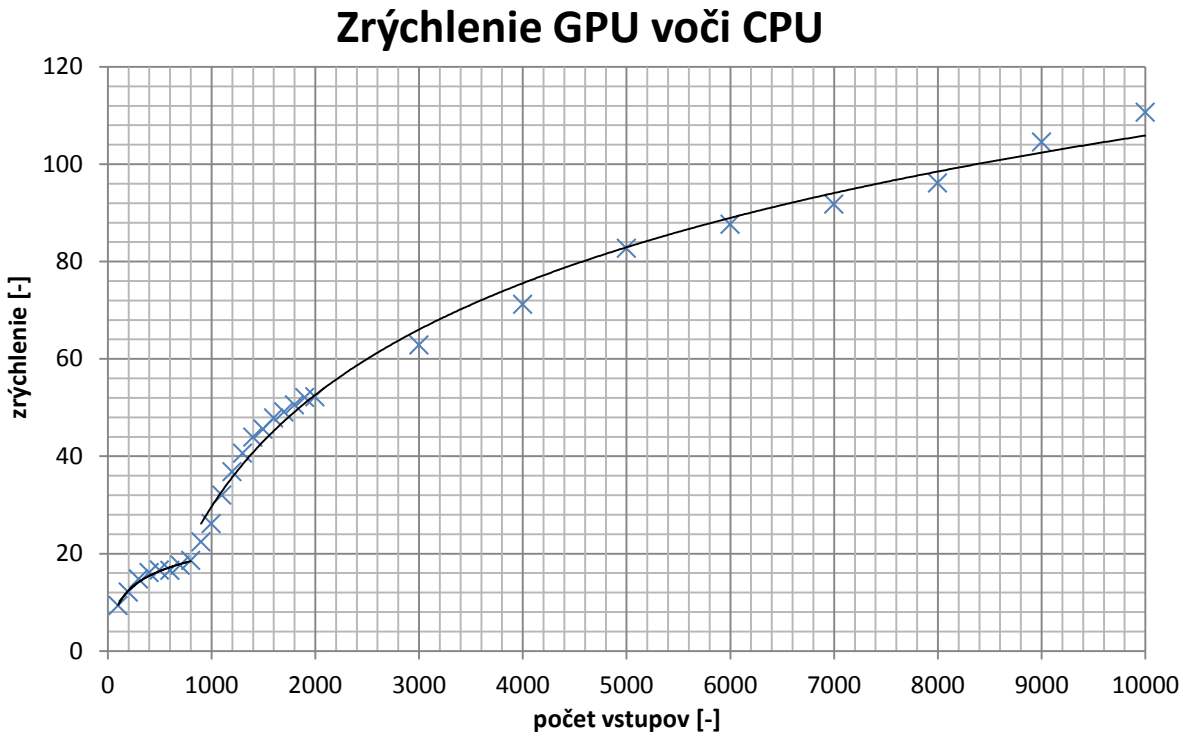
Graf z Obrázok 14: Škálovanie výkonu CPU ukazuje, že škálovanie procesora je lineárne, s dvoma bodmi zlomu. Je vidieť, že najskôr procesor škáluje veľmi dobre, kedy zdvojnásobenie počtu vstupov prináša takmer presne dvojnásobný čas potrebný pre výpočet. Avšak v rozmedzí 800 a 900 vstupov dochádza k zlomu, od ktorého sa škálovanie zhoršuje na

menej ako polovicu, kedy zdvojnásobenie vzoriek znamená takmer 4,25-násobok času potrebného na výpočet. Toto je spôsobené zahltením L1 cache pamäte procesora, ktorý je kvôli väčšiemu množstvu dát nútený po ich nenájdení v L1 cache čakať na ich dodanie z L2 cache. Následne, medzi počtom vstupov 3000 a 4000, dochádza k ďalšiemu zlomu, kedy už procesoru nestačí ani L2 cache a je nútený používať na medzivýpočty pamäť RAM. Škálovanie znovu klesá približne na tretinu, kedy zdvojnásobenie vzoriek znamená v priemere trojnásobný čas. Týmto celkové škálovanie pri vysokom počte vstupov klesá z počiatočných viac ako 98% na 16%, čo je nevyhovujúce, a daná neurónová sieť je bez ďalších úprav nevhodná na trénovanie pomocou CPU pri vysokom počte vstupov.



Obrázok 15: Škálovanie výkonu GPU pri 1000 epochách

Graf z Obrázok 15: Škálovanie výkonu GPU pri 1000 epochách ukazuje lineárne škálovanie na celej testovanej množine, kedy zdvojnásobenie počtu vzoriek znamená zvýšenie potrebného času o približne 55%, čo je možné považovať za ideálne škálovanie. Toto je spôsobené nie 100% vyťažením GPU, ktoré sa nepodarilo dosiahnuť ani pri použití 10000 vstupov (program MSI Afterburner vypočítal vyťaženie 98% už pri 4000 vstupoch, avšak ani ďalšie zvýšenie vstupov na 10000 túto hodnotu neprekročilo). Je možné predpokladať, že ďalším pridávaním vzoriek by došlo k plnému vyťaženiu grafickej karty, čo by spôsobilo zlom a výrazné zhoršenie škálovania.



Obrázok 16: Zrýchlenie učenia neurónovej siete pri použití GPU

Graf z Obrázok 16: Zrýchlenie učenia neurónovej siete pri použití GPU ukazuje, že v rozpätí 400 až 600 vstupov došlo k ustabilizovaniu zrýchlenia na hodnote 16,5. Avšak následne procesoru nedostačovala dátová priepustnosť cache pamäte a došlo k opätovnému rastu zrýchlenia učenia neurónovej siete pomocou GPU, ktoré narástlo až na hodnotu 110x pri 10000 vstupoch. Taktiež je vidieť, že zrýchlenie má približne logaritmickú závislosť, čo poukazuje na postupné limitné približovanie sa počtu vstupov, pri ktorom už nebude dochádzať k nárastu zrýchlenia. Táto situácia nastane v momente, kedy dôjde k 100% vyťaženiu procesora.

10 ZLOŽITOSŤ IMPLEMENTÁCIE PROGRAMU PRE GPU

Implementované algoritmy, ako je možné vidieť v prílohe so zdrojovými kódmi, ukazujú iba drobné odlišnosti medzi verziami pre CPU a GPU. Z toho sa dá usudzovať, že pri zvolení vhodného problému (dostatočná miera paralelizácie, ideálne nezávislosť na opakovanom prenose dát medzi CPU a GPU a malom výskyte vetvenia kódu) je prechod na GPU implementáciu algoritmu relatívne jednoducho realizovateľný. V prípade, že sa s použitím GPU počíta už od začiatku, je implementovanie ešte časovo jednoduchšie, nakoľko pri implementovaní je vhodné najskôr vyvinúť a otestovať verziu pre procesor v jazyku C podľa štandardu C99, s vyvarovaním sa prvkov, ktoré GPU nepodporuje.

Optimalizovanie programov pre GPU je už však náročnejšie a vyžaduje pochopenie architektúry každého výrobcu. Taktiež je naň vhodné myslieť už pri vývoji testovacej CPU verzie, čím dochádza k skráteniu celkového času potrebného na vývoj OpenCL verzie programu (je nutná menšia, poprípade žiadna, miera refaktoringu existujúceho kódu).

Pri písaní hostiteľskej verzie kódu je nutné dbať na kontrolu chýb pri volaní každej OpenCL funkcie, kedy jej zlyhanie sa môže prejaviť o desiatky až stovky riadkov kódu neskôr, v najhoršom prípade ako zdanlivá chyba v kerneli, ktorá nemá zjavnú príčinu a CPU ekvivalent programu funguje bez najmenších problémov. Pre možnosť použitia programu na grafických kartách od rôznych výrobcov je nutné nielen presne dodržiavať štandard, ale byť si vedomý odchýlok od neho u každého výrobcu, ktorého produkty budú použité. AMD ani NVIDIA ho totižto neimplementujú v plnom rozsahu, čo môže spôsobiť, že program bežiaci na kartách jedného výrobcu môže na GPU druhého výrobcu produkovať nesprávne výsledky, prípadne sa nemusí vôbec spustiť.

Ako príklad slúži firma NVIDIA, ktorá povoľuje pri alokácii pamäti pre GPU, ktorá sa nenapĺňa dátami, špecifikovať inicializačnú premennú. Toto však je v priamom rozpore so štandardom, podľa ktorého je takáto operácia neplatná a pamäť sa nealokuje.

Za samozrejmosť je možné považovať nutnosť rátať s rôznymi parametrami možných grafických kariet, a tieto pri vývoji aplikácie zohľadniť.

Taktiež je nutné dávať pozor na veľkosti dátových typov, ktoré nemusia mať na CPU rovnakú veľkosť ako na GPU. Z tohto dôvodu je vhodné pre dáta, ktoré sa budú kopírovať medzi zariadeniami, používať na hostiteľskom zariadení dátové typy definované v OpenCL (obsahujú predponu `cl_`).

Z dôvodu nereagovania GUI počas výpočtu a následného automatického obnovenia ovládačov je vhodné plne nezaťažovať GPU, alebo využiť zostavu, v ktorej je na zobrazovanie použitá iná grafická karta. Nevýhodou programovania GPU sú výrazné obmedzenia (za najväčšie považujem nemožnosť dynamickej alokácie pamäte na zariadení za

behu programu, ďalšie sú uvedené v kapitole 5: Výhody a nevýhody GPGPU), ktoré sťažujú vývoj programov. Ďalej je to relatívne veľká základná réžia na prístup k OpenCL zariadeniu, ktorá v prípade použitia bežného CPU odpadá. Taktiež pre dosiahnutie maximálneho výkonu je nutné plné porozumenie princípu fungovania architektúry, ktorá sa bude využívať (vyvarovanie sa konfliktov prístupu do pamäte, zbytočnej serializácii programu). Toto všetko predlžuje čas vývoja aplikácie.

11 ZÁVER

Táto bakalárska práca rozoberala možnosti distribúcie výpočtov v PC pomocou grafických procesorov a nástroja OpenCL.

Úvod sa zaoberal postupným prechodom k multijadrovým procesorom a vznikom grafických kariet s podporou GPGPU (kapitola 2: Zvyšovanie výkonu), kde sú popísané výpočtové možnosti súčasnej generácie grafických kariet. Následne boli rozobrané spôsoby paralelizácie výpočtov, ich obmedzenia a problémy, ktoré pri nich vznikajú (kapitola 3: Paralelizácia). Tým bol splnený prvý bod zadania.

Kapitoly 4 a 5 sa zaoberali využitím GPGPU a výhodami a nevýhodami GPU, čím bol splnený tretí bod zadania.

Následne, v kapitole 6: Vývojové nástroje pre GPGPU, boli postupne vymenované tri najpoužívanejšie nástroje (CUDA, OpenCL a DirectCompute). Pri každom boli špecifikované jeho možnosti a obmedzenia. Tým bol splnený druhý bod zadania. Na základe týchto skutočností bol v kapitole 7 zvolený ako vývojový nástroj OpenCL (multiplatformový z hľadiska hardwaru aj softwaru, otvorený štandard). Následne došlo k rozobratiu jeho základných princípov, pamäťových priestorov a prístupov k pamäti. Tým bol splnený štvrtý bod zadania. Taktiež boli popísané princípy optimalizácie a spôsob tvorby jednoduchej aplikácie, čím bol začatý piaty bod zadania.

Kapitola 8 demonštrovala výkonnosť grafickej karty implementovaním MD5 algoritmu na hľadanie hesla hrubou silou z MD5 šifry. Zároveň bol daný algoritmus paralelizovaný aj na procesore a aby bolo možné výsledky porovnať.

Prelomenie hesla "zzzzzzz" o dĺžke 7 znakov z jeho MD5 šifry trvalo na procesore Intel Core 2 Duo Q6600 pri štarte od reťazca "aaaaaaa" 313,74 sekúnd, v prípade grafickej karty NVIDIA 560Ti-448 to však bolo iba 5,71 sekúnd, čo je zrýchlenie približne 55-krát. Ukázalo sa, že prechod z CPU na GPU prináša v tomto prípade približne 45-násobné zrýchlenie pri uvažovaní hardwaru v rovnakej cenovej hladine (viď teoretický výkon v Tabuľka 12: Výkonnostné porovnanie CPU a GPU pre MD5 algoritmus). Možnosťou takéhoto zrýchlenia došlo k výraznému zníženiu bezpečnosti šifrovacích algoritmov, hlavne v prípade krátkych hesiel, ku ktorým je možné získať kontrolnú sumu. Táto bezpečnosť ďalej klesá s možnosťou distribúcie výpočtov medzi viacero procesorov.

Tým bol dokončený a splnený piaty bod zadania a zároveň začatý siedmy bod zadania.

Kapitola 9 popisuje realizovanie komplexnejšej úlohy, kedy bola implementovaná neurónová sieť na rozpoznávanie ručne písaných číslic (OCR) s možnosťou učenia. GPU verziu algoritmu bola taktiež porovnaná s paralelizovanou CPU verziou. V tomto prípade už bola hodnota zrýchlenia nižšia, čo bolo spôsobené zložitejším výpočtom a nižšou mierou

paralelizácie. Taktiež réžia nutná na opakované volanie každého kernela zaberá určitý čas a tým znižuje možnosť zrýchlenia.

Výsledky ukazujú, že zrýchlenie je v prípade rovnakej cenovej hladiny 13 - násobné. Merania z kapitoly 9.8: Podrobné porovnanie škálovania výkonu CPU a GPU ukazujú, že táto hodnota môže ešte rásť v závislosti od počtu vstupov. Tieto hodnoty na prvý pohľad nie sú závratné, avšak pri učení zložitých neurónových sietí vyžadujúcich vstupné množiny s tisícami prvkov a k tomu ešte tisícmi epoch, dochádza k výraznému šetreniu času. Grafy ukázali, že už pri 1000 epochách pri 1000 vstupných vzorkách je v absolútnych číslach čas potrebný procesorom Intel Core 2 Duo Q6600 994 sekúnd, avšak grafickou kartou NVIDIA 560Ti-448 je to iba 38 sekúnd. Toto môže v konečnom dôsledku znamenať skrátenie niekoľkodňového učenia na pár hodín, a k tomu výraznú úsporu energie.

Táto kapitola taktiež ukázala, že u CPU je výrazne jednoduchšie dosiahnuť kritickej hodnoty, kedy dátová cache procesora neposkytuje dostatočnú priepustnosť dát a dochádza k výraznému spomaleniu výpočtov. To pre grafickú kartu znamená ďalšie urýchlenie, ktoré sa pri teste 10000 vstupných čísel vyšplhalo až na hodnotu 110. Avšak ani pri tomto počte vstupov nedošlo k plnému využitiu potenciálu testovanej grafickej karty. Týmto bol splnený šiesty bod zadania a ďalšia časť siedmeho bodu.

V kapitole 10 sú zhodnotené skúsenosti pri implementácii algoritmov pre GPU. Ukázalo sa, že v prípade plánovaného využitia grafickej karty na výpočty a pri zvolení vhodnej úlohy nie je implementácia pre GPU výrazne náročnejšia ako implementácia iba štandardnej CPU verzie. Avšak pre dosiahnutie maximálneho výkonu je nutné plné porozumenie princípu fungovania architektúry, ktorú budeme využívať (vyvarovanie sa konfliktov prístupu do pamäte, zbytočnej serializácii programu), čo predlžuje čas vývoja aplikácie. Touto kapitolou bol zároveň dokončený siedmy bod zadania.

Vzhľadom k tomu, že teoretický výkon grafických kariet rastie výrazne rýchlejšie ako u procesorov (viď Tabuľka 12: Výkonnostné porovnanie CPU a GPU pre MD5 algoritmus), dá sa do budúcnosti očakávať, že tento pomer sa bude ďalej zvyšovať.

Na základe výsledkov nameraných v kapitolách 8 a 9 je možné jednoznačne odporučiť použitie grafických procesorov pri výpočtoch, ktoré sa dajú masívne paralelizovať a nie sú príliš pamäťovo náročné (napríklad neurónové siete, spracovanie obrazu, kódovanie a dekódovanie videa a audia).

Programy testované v tejto bakalárskej práci je možné ďalej rozšíriť a pokračovať tak vo vývoji priemyselných, prípadne komerčných programov, umožňujúcich hromadnú identifikáciu výrobkov, ich sériových čísiel alebo rozpoznávanie textov. Taktiež je možné v ďalších prácach, či už bakalárskych alebo diplomových, pokračovať štúdiom problémov načrtnutých pri ich implementácii, ako je rozšírenie neurónovej siete na rozpoznávanie abecedy, segmentácia písma, distribúcia výpočtov na viacero procesorov alebo optimalizácia programov pre GPU. Týmto je splnený ôsmy bod zadania.

12 LITERATÚRA

[1] CHANDRAKASAN, A. P. et al. Optimizing Power Using Transformations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 1995, č. 14, s. 12–31. ISSN 0278-0070.

[2] HARRIS, M. *Mapping Computational Concepts to GPUs*. 2. 2005 [cit. 2013-05]. ISBN 0-321-33559-7. Dostupné z: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter31.html

[3] NVIDIA. NVIDIA GeForce 8800 GPU Architecture Overview. *NVIDIA.COM* [online]. 2006, verze November 2006 [cit. 2013-04]. Dostupné z: http://www.nvidia.com/object/IO_37100.html

[4] CHAPMAN, B. G. JOST a R. V. D. PAS. *Using OpenMP: Portable Shared Memory Parallel Programming*. London, England: The MIT Press, 2007. ISBN 978-0-262-53302-7. Dostupné také z: http://scc.qibebt.cas.cn/docs/program/openmp/MIT.Press.Using.OpenMP.Portable.Shared.Memory.Parallel.Programming.Oct.2007.pdf

[5] AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: *AFIPS spring joint computer conference*. Atlancic City: 1967, s. 483-85.

[6] GUSTAFSON, J. L. G. R. MONTRY a R. E. BENNER. Development and analysis of scientific application programs on a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Computing*. New Mexico: 1988, č. 4, s. 609-38. SAND 88-0317.

[7] BERNSTEIN, A. J. Analysis of Programs for Parallel Processing. *IEEE TRANSACTIONS ON ELECTRONIC COMPUTERS*. 1966, č. 5, s. 757-63. ISSN 0367-7508.

[8] NVIDIA. GPU-accelerated applications. *NVIDIA.COM* [online]. 2012, verze Október 2012 [cit. 2013-04]. Dostupné z: http://www.nvidia.com/docs/IO/123576/nv-applications-catalog-lowres.pdf

[9] ANDERSON, M. et al. Considerations When Evaluating Microprocessor Platforms. [online]. 2010 [cit. 2013-03]. Dostupné z: http://static.usenix.org/event/hotpar11/tech/final_files/Anderson.pdf

[10] KHRONOS. OpenCL. *khronos.org/opencl* [online]. 2012 [cit. 2013-05]. Dostupné z: http://www.khronos.org/opencl/

[11] NVIDIA. GPU Computing Webinars Introduction To OpenCL. *NVIDIA.COM* [online]. 2009, verze 2009 [cit. 2013-03]. Dostupné z: http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_Introduction_To_OpenCL.pdf

[12] NVIDIA. OpenCL Best Practices Guide. *NVIDIA.COM* [online]. 2009, verze August 2009 [cit. 2013-05]. Dostupné z: http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf

[13] MUNSHI, A. et al. *OpenCL Programming Guide*. Boston: Addison-Wesley, 2012. ISBN 978-0-321-74964-2.

[14] RIVEST, R. RFC 1321 (MD5 algorithm). In: *IETF Tools* [online]. 1992 [cit. 2012-11]. Dostupné z: http://tools.ietf.org/html/rfc1321

[15] HWSK S.R.O. Cenový prehľad. In: *pc.zoznam.sk* [online]. 2012. Dostupné také z: http://pc.zoznam.sk/clanky/cenovy-prehlad

[16] O'NEIL, M. Neural Network for Recognition of Handwritten Digits. In: *codeproject.com* [online]. 2006 [cit. 2013-04]. Dostupné z: http://www.codeproject.com/Articles/16650/Neural-Network-for-Recognition-of-Handwritten-Digi

[17] STERGIOU, C. a D. SIGANOS. NEURAL NETWORKS. In: *Imperial college London* [online]. 1996 [cit. 2013-05]. Dostupné z: http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html

[18] MADHUSUDANAN, A. Designing And Implementing A Neural Network Library For Handwriting Detection, Image Analysis etc.- …. In: *codeproject.com* [online]. 2009 [cit. 2013-03]. Dostupné z: http://www.codeproject.com/Articles/14342/Designing-And-Implementing-A-Neural-Network-Librar

13 ZOZNAM PRÍLOH

Príloha 1: Slovník pojmov

Príloha 2: Namerané výsledky (na DVD)

Príloha 3: Zdrojové kódy (na DVD)

Príloha 4: Aplikácie kompilované pre platformu Windows (na DVD)

Príloha 5: Elektronická verzia práce (na DVD)

PRÍLOHA 1: SLOVNÍK POJMOV

CPU (Central Processing Unit) – centrálna riadiaca jednotka (procesor). Je to primárna výpočtová jednotka počítačového systému. Prebiehajú tu všetky matematicko-logické operácie. Procesor úzko komunikuje s operačnou pamäťou, ktorá ho zásobuje dátami na spracovanie.

GPU (Graphics Processing Unit) – grafická riadiaca jednotka (grafická karta). Prebiehajú tu špecializované matematicko-logické operácie, predovšetkým výpočty objektov v 3D priestore

Shaders – Špeciálne výpočtové jednotky v jadre GPU. Delíme ich na 3 základne typy:

- Vertex Shader – v minulosti používaný na vykresľovanie nerovností v hrách
- Pixel Shader – v minulosti používaný na vykresľovanie dodatočných efektov ako tiene, explózie, odlesky, rozmazanie obrazu pri pohybe a podobne
- Unified Shader – používane v súčasnosti na všetky predchádzajúce úlohy, môžu vykonávať akúkoľvek funkciu

Textúrovacia jednotka – zabezpečuje vykresľovanie textúr v obraze

TnL jednotka – Transform and Lightning – zisťuje potrebnú komplexnosť objektov a zabezpečuje ich realistickosť

GPGPU (General Purpose Graphics Processing Unit) – využitie grafickej karty na všeobecné výpočty

Architektúra – typ konštrukcie mikroprocesorov, napr. x86 u CPU

Multithreading – Všeobecné označenie pre schopnosť programu vetviť samého seba na tzv. vlákna (threads), ktoré môžu pracovať súčasne a skrátiť tak dobu výpočtu

Multitasking – Všeobecné označenie pre schopnosť (najčastejšie operačného systému) pracovať na niekoľkých úlohách súčasne

Multiprocessing – Všeobecné označenie pre využitie viacerých procesorov pri práci na viacerých úlohách. Rozdelenie úloh môže byť:

- Symetrické – práve voľný procesor sa môže zapojiť do riešenia práve prebiehajúceho výpočtu
- Asymetrické – každý procesor má pevne pridelenú prácu napr. GPU s Pixel a Vertex shadermi

FLOPS – tiež flop/s – Float Point Operations Per Second – počet operácií s pohyblivou desatinou čiarkou za jednu sekundu. Obvykle sa touto jednotkou udáva výkon CPU a GPU.

Multi-core – Viac jadier v CPU alebo GPU – napr. štvorjadrový procesor

Multi-procesor – Viac CPU alebo GPU – napr. dve grafické karty v jednom počítači

Thread (vlákno) – najmenšia postupnosť inštrukcií, ktoré môžu byť nezávisle spracované operačným systémom

SIMD (Single instruction, multiple data) – architektúra, kedy sa nad určitým súborom dát zároveň prevádza jedna inštrukcia

SIMT (Single instruction, multiple thread) – v princípe SIMD, nad každými dátami beží vlastné vlákno

API – Je to špecifikácia rozhrania medzi rôznymi softwarovými komponentmi. V princípe ide o funkciu, ktorá okrem prototypu poskytuje aj vysvetlenie funkčnosti, vstupných parametrov a návratovej hodnoty.

Kernel – funkcia, ktorá bude vykonaná paralelne na zariadení, slúži ako vstupná funkcia programu a ako jediná môže byť volaná hostiteľom

HLSL (High Level Shader Language) – Proprietárny programovací jazyk pre programovanie efektov charakterizujúcich povrch objektov, vodu a podobne. V princípe sa jedná o distribúciu svetla na objekty v 3D grafickom priestore.

GUI (Graphical User Interface) – grafické používateľské rozhranie, umožňuje ovládať elektronické zariadenie pomocou interaktívnych obrazových prvkov

Race conditions – jav nastávajúci pri paralelizácii, kedy k zápisu do jednej premennej (jej inkrementácii) pristupuje zároveň viacero vláken, čím sa jej hodnota zvýši iba o hodnotu zapísanú ako poslednú

Atomická operácia – operácia, ktorá zabraňuje race conditions serializáciou programu