Jihočeská univerzita v Českých Budějovicích

Přírodovědecká fakulta

Bakalářská práce

Výpočty na grafických kartách

Aleš Svoboda

školitel: RNDr. Milan Předota, Ph.D.

České Budějovice 2012

Bibliografické údaje
Svoboda A. 2012: Výpočty na grafických kartách.
[Computation on Graphics Cards. Bc.. Thesis, in Czech.] – 54 p., Faculty of Science, The University of South Bohemia, České Budějovice, Czech Republic.

Anotation:
Purpose of this work is to show possibility of using graphics card not just for 3d graphics but also for computations which were for a long time just a domain of CPU. In the first part we are looking at what graphics card is as well as APIs used for their programming. Second part shows us OpenCL API on an example. Last part is measurement of performance between CPU and GPU.

Anotace:
Cílem této práce je ukázat možnost používat grafickou kartu nejen pro 3D grafiku, ale také pro výpočty, které byli dlouhou dobu doménou pouze CPU. V první části se dozvídáme co je to vlastně grafická karta a také jaké API se využívají pro jejich programování. Druha část nám ukazuje už samotné OpenCL na příkladu. Poslední část je porovnání výkonu mezi CPU a GPU.

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě – v úpravě vzniklé vypuštěním vyznačených částí archivovaných Přírodovědeckou fakultou - elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

25.4.2012 ….....

Obsah

1.Úvod....1
1.1 Kde se využívá grafická karta....1
1.2 Cíl práce....1
2 Grafická karta....1
2.1 Charakteristika....1
2.2 Komponenty grafické karty....2
2.2.1 GPU....2
2.2.2 Paměť....2
2.2.3 Bios....2
2.2.4 RAMDAC....2
2.2.5 Výstupy....2
3 GPGPU....3
3.1 CPU vs GPU....3
3.2 CUDA....3
3.2.1 Popis....3
3.2.2 Architektura....4
3.2.3 Paměťový model....5
3.2.4 Výhody a nevýhody....6
3.3 AMD Stream....7
3.4 OpenCL....7
3.4.1 Popis....7
3.4.2 Architektura....8
3.4.3 Optimalizace....11
3.4.4 Výhody a nevýhody....11
3.5 DirectCompute....12
4 Návrh počítače pro výpočty pomocí GPU....12
4.1 Procesor....12
4.2 Základní deska....12
4.3 Grafická karta....13
4.4 Paměť....13
4.5 Zdroj....14
4.6 Harddisk....14
4.7 Skříň....14
4.8 Mechanika....14
5 Příprava počítače....15
5.1 Nastavení Microsoft Windows 7....15
5.1.1 Instalace....15
5.1.2 Ovladače....15
5.1.3 Visual Studio....16
5.1.4 Cuda Toolkit....16
5.1.5 Nastavení projektu ve visual studio pro OpenCL....16
5.2 Nastavení OpenSuse....19
5.2.1 Instalace....19
5.2.2 Ovladače....20
5.2.3 Kompilátor....20
5.2.4 Cuda Toolkit....20
5.2.5 Kompilace projektů používajících OpenCL....20
6 Ukázka OpenCL na příkladu....21
6.1 Host soubor....21
6.2 Kernel....29

6.3 Další zajímavé příkazy....30
7 Porovnání rychlosti....30
7.1 Specifikace počítače....30
7.2 Násobení matic....31
7.5 Transponování matice....31
8 Závěr....32
Seznam použité literatury....33
Seznam příloh....34
A Zdrojové kódy....35
A.1 OpenCL – Násobení Matic....35
A.2 Násobení matic v C....40
A.3 OpenCL – Transponování matic....42
A.4 Transponování matice v C....47
B Obsah CD....48

1.Úvod

1.1 Kde se využívá grafická karta

Grafické karty po dlouhou dobu sloužili pouze k zobrazování obsahu. Postupně ale docházelo k jejich postupnému vylepšování až to dospělo do stadia, kdy se dají využít i na jiný účel než jen grafické operace. Tato schopnost se nazývá GPGPU neboli general-purpuse computing on graphics processing unit. Cílem je nahradit CPU u operací kde je GPU mnohem lepší volbou popřípadě podpořit CPU pomoci grafické karty.

V současné době se už objevují programy využívající tyto nové schopnosti grafických karet pro vlastní urychlení. Ale zdaleka nejvíce se začínají používat u počítačů, jež jsou určeny pro výpočty, simulace a další. Tyto počítače musí být schopny počítat velmi rychle, proto se využívá kombinace GPU a CPU. Další oblastí kde dochází pomalinku k využívání GPU jsou tzv distribuované výpočty z těch jsou nejznámější SETI@home a např Folding@home kdy se do nich může zapojit kdokoliv s počítačem a připojením k internetu.

1.2 Cíl práce

Cílem teto práce je seznámení se s problematikou programovaní GPU, včetně výběru vhodného hardware a zároveň i instalace potřebného softwarového vybavení včetně operačního systému (Linux a Windows). Dále vytvoření několika malých programu a pomocí jednoho z nich vysvětlení programování GPU. Jako poslední se provede pomocí těchto malých programu srovnání rychlosti CPU a GPU v určitých operacích.

2 Grafická karta

2.1 Charakteristika

Jedná se o součást počítače ,jenž se stará o zobrazení obrazu na monitoru popřípadě TV a dalších. Některé z nich umožňují i připojení vstupu např z videokamery. V oblasti grafických karet určených pro domácnost jsou nejznámější grafické karty od AMD a Nvidia.

2.2 Komponenty grafické karty

2.2.1 GPU

Jedná se o speciální procesor, který je optimalizován pro výpočty v plovoucí desetinné čárce. Využití nachází v renderování 3D grafiky a v masivních paralelních výpočtech. Hlavní vlastností rozlišující jednotlivá GPU je jejich architektura,frekvence čipu a počet výpočetních jednotek.

2.2.2 Paměť

Jak již napovídá název paměť slouží k ukládání informací, které jsou potřebné k výpočtům. Její velikost se pohybuje od 1 MB v případě velmi starých karet až po 8 GB u speciálních karet. Stejně jako v případě GPU i tady záleží na frekvenci a architektuře. Často je mnohonásobně rychlejší než operační paměť počítače.

2.2.3 Bios

Obsahuje základní program, který je zodpovědný za funkci grafické karty. Zároveň obsahuje informace o časovaní paměti i GPU, jejich napájení a dalších. Stejně jako v případě biosu základní desky lze BIOS updatovat, ale jedná se o mnohem rizikovější operaci než v případě základní desky.

2.2.4 RAMDAC

Používá se pro převedení digitálních signálu na analogový, aby se ten dal zobrazit na CRT monitorech. V současné době se už moc nevyužívá, většina novějších rozhraní přepravuje signál digitálně, ale stále se montuje na grafické karty.

2.2.5 Výstupy

Zřejmě nejznámější je výstup VGA používaný pro připojeni CRT monitoru. S příchodem LCD technologie došlo k přechodu na DVI rozhraní a v dnešní době se začíná prosazovat HDMI rozhraní. Další z možných konektoru je S-Video.

3 GPGPU

3.1 CPU vs GPU

CPU a GPU se vyvinuli každý pro jiný účel. CPU slouží jako výpočetní jednotka a také se využívá k obsluze ostatních zařízení pomocí přerušení. Cílem CPU je zpracovat jednu operaci jak nejrychleji to jde. GPU je specializovaný hardware pro renderování grafiky. Takže GPU musí být schopno zpracovat operaci pro co největší počet dat za co nejméně času. Výsledkem je že GPU poskytuje velké zvýšení výkonu v určitých operacích, obsahuje velké množství SIMD procesorů a má nativní podporu pro paralelismus zabudován v hardware. Na druhou stranu CPU poskytuje uspokojivý výkon ve všech operacích, ale nemá zase takovou podporu pro paralelismus. Víz Obrázek 1.[1]

Obrázek 1: Rozdíl mezi CPU a GPU[zdroj: Nvidia]

3.2 CUDA

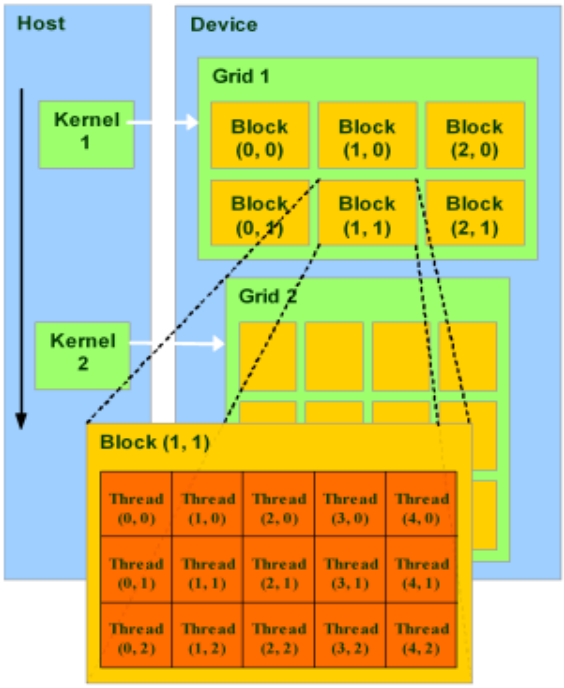
3.2.1 Popis

CUDA(Compute Unified Device Architecture) je architektura, která byla představena v roce 2006 a byla vyvinuta za účelem využití grafických karet Nvidia pro klasické výpočty. Můžeme o ní uvažovat jako o schematu podle kterého Nvidia vytváří grafické karty schopné kromě grafických operací i klasických výpočtu. První takovouto kartou byla Geforce 8800GTX představená v témže roce. V roce 2007 poté vyšla první verze SDK. Později byla přidána i možnost výpočtu v double precision a také využití více grafických karet. Tato architektura nabízí podporu API jako např Direct Compute, OpenCL, CUDA C. Na host

soubor se také muže použít velké množství jazyků jako např Java,C++,C a další. Jako první se pro kernel využívala tzv CUDA C , ale s příchodem dalších podporovaných technologii to už není nutné.

3.2.2 Architektura

Je definována na tzv kernel – host modelu. Kdy kernel je ta část uživatelem definované funkce, která je zpracovávána na zařízení paralelně jednotlivými vlákny(thread). Každé Nvidia GPU obsahuje nějaké množství Streaming Multiprocessors, které mohou zpracovávat velké množství vláken dohromady. Zpracování vláken je řízeno automaticky tzv Hardware thread manager. Vlákna jsou dále řazena do bloku(Blocks) o 1,2,3 dimenzí, které jsou zase součástí jedno popřípadě dvou-dimenzionální mřížky(grid) viz Obrázek 2.



Obrázek 2: CUDA Model [zdroj: Nvidia]

3.2.3 Paměťový model

Na grafických kartách Nvidia můžeme najít až 6 druhu paměti[2]:

- **Pole registru**

  Každé vlákno má svůj registr ke kterému může přistupovat. Jednotlivé registry jsou umístěny na svém vlastním stream multiprocesoru

- **Lokální paměť**

  Pokud dojde k vyčerpání registru začne se využívat tato paměť. Vzhledem k tomu že jde o náhradu registrů, tak se chová podobně kdy každé vlákno má svojí lokální paměť . Bohužel ale je tato paměť pomalejší než registr. Důvodem je její umístění v globální paměti.

- **Sdílená paměť**

  Jedná se o paměť, která je přístupna všem vláknům v daném bloku. Vzhledem k tomu že se jedná o paměť, která je umístěna přímo na čipu není problém s její rychlostí. Poskytuje možnost komunikace mezi jednotlivými vlákny a je kontrolována přímo vývojářem.

- **Globální paměť**

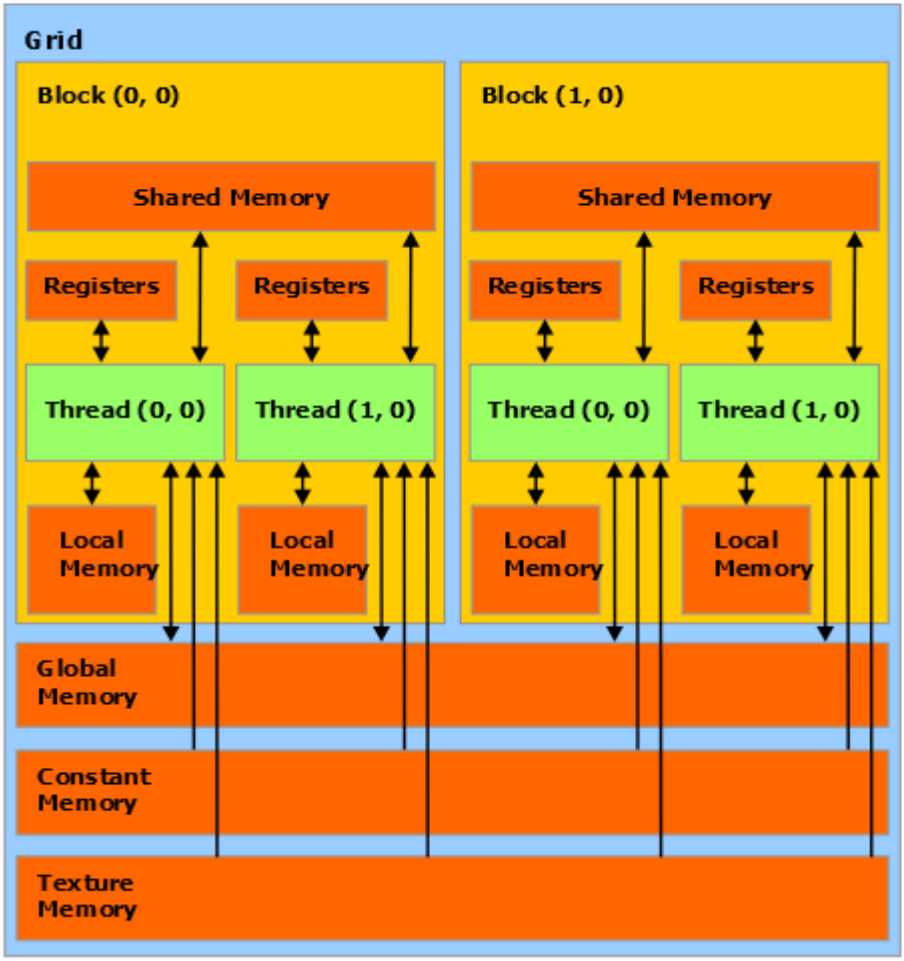
  Největší paměť použitelná všemi streaming multiprocesory, od 256 MB až po 1.5 GB u moderních grafických karet (4 GB u speciálních Tesla karet). Poskytuje poměrně velkou rychlost přenosu (100 GB/s), ale trpí poměrně velkými latencemi(několik stovek cyklů). Nelze ji cachovat

- **Konstantní paměť**

  Jedná se o paměť, která je dostupná pouze pro čtení. Stejně jako v případě Globální paměti je sdílená všemi streaming multiprocesory, ale na rozdíl od globální je přítomna v L1 cache multiprocesoru. Je také docela pomalá.

- **Texturová paměť**

  Je dostupná ke čtení všem multiprocesorům. Data jsou posílána skrze texturovací jednotky GPU. Optimalizována pro 2D prostorovou lokalitu. Stejně pomalá jako globální paměť.



Obrázek 3: CUDA Paměťový model[zdroj: Nvidia]

3.2.4 Výhody a nevýhody

Jako jedná z výhod rozhraní CUDA C se může jevit velké množství existujících vyřešených příkladů a také hodně literatury, která se dá koupit popřípadě stáhnout.

Další výhodou muže být dobrá podpora společností Nvidia a zároveň se často ukazuje že CUDA C je rychlejší než konkurenční řešení (často se ale rozdíl zmenší pokud dojde k dobré optimalizaci).

Bohužel jedná z nevýhod CUDA C je závislost pouze na jednom výrobci grafických karet konkrétně Nvidia, kdy program který uděláme nespustíme na konkurenční značce.

Další z nevýhod se projevuje v souvislosti s operačním systémem Linux, kdy

kompilátor jazyka CUDA C potřebuje specifickou verzi GCC a GlibC aby fungoval. Často se stane že doinstalování starší verze GCC je do novějších verzí linuxu problém, ale není neřešitelný.

3.3 AMD Stream

Jiný název by mohl být AMD App Acceleration. Jedná se o platformu, která přišla společně s kartami ze série Radeon X1xx. Tyto karty jako první měli k dispozici dostatečně programovatelné shadery, které se dali využít i na další věci než jen 3D grafiku. Jako první rozhraní se v tomto ohledu používalo tzv Close to Metal, které nahradilo používání DirectX a OpenGL rozhraní. AMD později přešlo na podporu OpenCL u novějších karet.

3.4 OpenCL

3.4.1 Popis

OpenCL(Open Computing Language) je otevřený standard, který umožňuje paralelní programování heterogenních počítačových systému. OpenCL specifikace definuje standard, který poté zařízení musí implementovat pokud chce využít možností OpenCL. Je podporovaný jak data a task paralelní programovací model.

Programování OpenCL se skládá ze 2 částí. V první části máme tzv host soubor, který slouží pro zjištění zařízení, jejich nastavení a určení na jakém z nich by mělo dojít ke spuštění. Další častí je programování tzv kernelu, což je samotná funkce, která se nám má spustit na zařízení. Programování kernelu probíhá v C99 dialektu programovacího jazyka C. Samotné programování má také určitá omezení, které se objevují vzhledem k určení tohoto jazyka. Jsou k dispozici i určitá rozšíření. Další zajímavou vlastností muže byt možnost spolupráce s OpenGL a dalšími grafickými API. Dále samotný host soubor nemusí být jen v C, je umožněno použití i jiných jazyku jako např Java, Python, C++ a další.

Samotný prvotní návrh OpenCL byl vytvořen firmou Apple. V Červnu 2008 byl poté přednesen Khronos konsorciu jako základ nového standardu. Skupina odpovědná za nový standard byla vytvořena ještě týž měsíc. Tato skupina obsahovala takové giganty jako AMD,Intel,IBM,Nvidia a další. Samotná specifikace poté byla vydána 8.Prosince 2008.

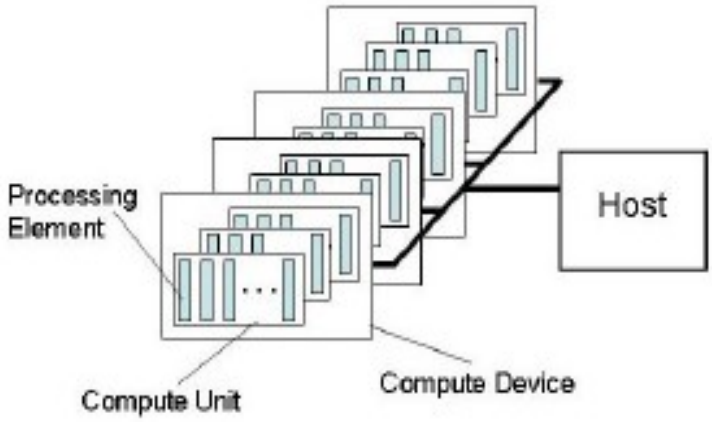
3.4.2 Architektura

Abychom mohli porozumět OpenCL je nutné zjistit jak samotna platforma pracuje, její paměťový model a samotné spouštění na zařízeních. Tyto věci najdeme v samotné specifikaci[3].

3.4.2.1 Model platformy

Vzhledem k tomu že samotná OpenCL platforma je heterogenní, což znamená že i když se architektura paralelních procesoru hodně liší, tak díky abstrakci s nimi stále můžeme pracovat.

Jak ukazuje obrázek 4 samotná platforma se skládá z hosta, který je připojen k jednomu popřípadě vice zařízením(Compute device). Program který běží na hostiteli pracuje se zařízeními pomocí API, které je poskytováno OpenCL standardem. To co se spouští na zařízeních je tzv kernel což je ta část programu, kterou často chceme paralelizovat. Samotné zařízení si můžeme představit jakou soubor výpočetních jednotek, které obsahují jeden nebo vice procesních elementu. Samotné zařízení muže představovat např grafická karta popřípadě vice-procesorový systém a další.



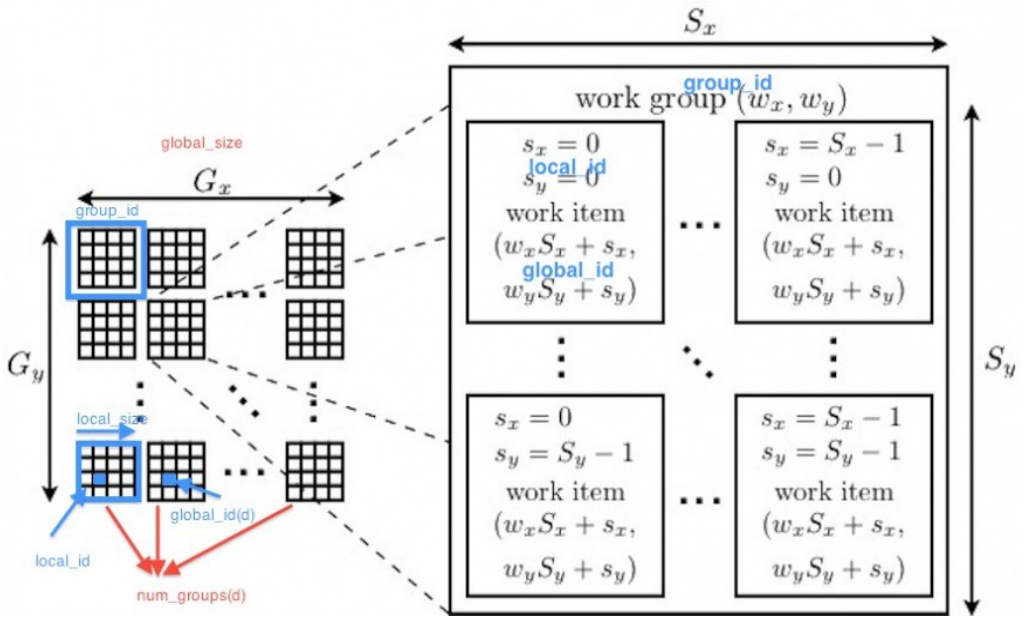
Obrázek 4: OpenCL model platformy

3.4.2.2 Vykonávací model

Spouštění OpenCL programu se dá rozdělit do 2 častí. Host část která se stará o nastavování zařízení a samotné spouštění kernelu na zařízení. Další častí je poté samotný kernel, který se spouští na jednom či více zařízeních.

Host program má za úkol zjistit si informace o platformě a zařízení. Tyto informace jsou poté využity pro vytvoření *contextu* což je prostředí, které slouží k samotnému spouštění kernelu. Context obsahuje informace o tom na jakých zařízeních má dojit ke spouštění, množství volné paměti, která je dostupná zařízením, dále samotnou správu paměti a harmonogram spouštění. Po tom co je context správně nastaven je možné vytvoření programu. Program samotný se skládá z jednoho či několika kernelu, konstant a občas i pomocných funkcí. Další částí je samotné linkování a kompilace. Pokud nechceme, aby se program kompiloval až po spuštění lze využít již zkompilovaný kernel. Po nastavení kernel parametru dojde k samotnému spuštění kernelu na zařízení.

Samotný kernel lze spustit v jedno-dimenzionálním,dvou-dimenzionálním nebo tří-dimenzionálním prostoru což nastavujeme před jeho spuštěním. Základní jednotkou v tomto prostoru je tzv vlákno (work-item). Každé vlákno zpracovává ten samý úsek kódu pro stejná popřípadě rozdílná data. Soustava vláken se stává pracovní skupinou (work-group). Vlákna v pracovní skupině pracují nezávisle na sobě, ale mají přístup ke společné pracovní skupinou sdílené paměti. Dále je možné je i synchronizovat speciálním příkazem. Každé vlákno má také svůj unikátní globální identifikátor (unique global ID). Další možností jak zjistit o jaké vlákno jde je použití lokálního ID, které identifikuje vlákno v jeho pracovní skupině a samotné unikátní ID pracovní skupiny.



Obrázek 5: model vykonávání[4]

3.4.2.3 Paměťový model

OpenCL paměťový model rozeznává 4 druhy paměti, které jsou k dispozici kernelu. Samotná paměť se liší v rychlosti přístupu, její dostupnosti v kernelu a hostu. Paměťový model je ukázán na Obrázku 6.

- **Globální paměť (Global Memory)**

  Jedná se o paměť, která je přístupná jak hostu tak kernelu k zápisu a čtení. Doba rychlosti přístupu je největší z dostupných. Přístup muže být nahrán do vyrovnávací paměti pokud to zařízení umožňuje.

- **Konstantní paměť (Constant Memory)**

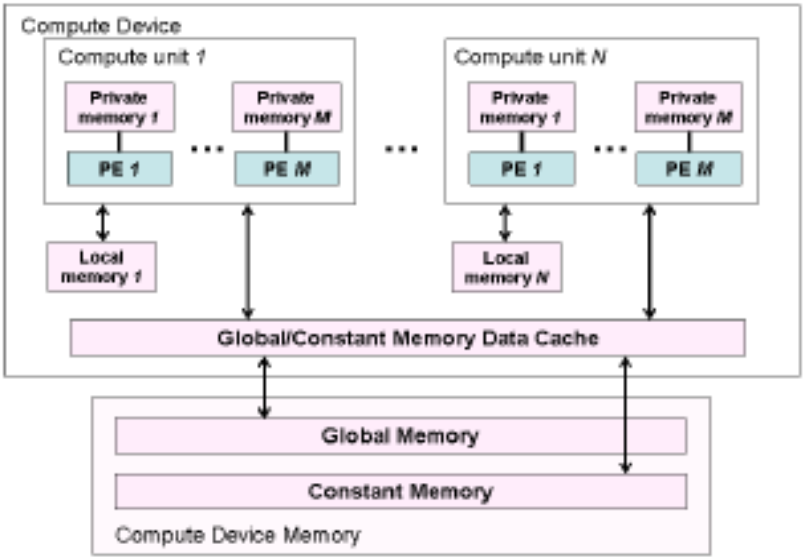
  Má v podstatě podobné vlastnosti jako globální paměť akorát data v ní uložena nemohou byt modifikována z kernelu. Rychlost přístupu by měla byt o něco menší než u globální paměti

- **Lokální paměť (Local Memory)**

  Tato paměť je viditelná pouze jednotlivým pracovním skupinám neboli Compute unit(work-group), kdy každá skupina má svojí unikátní. Může byt implementováno jako dedikovaná oblast paměti na zařízeních. Popřípadě lokální paměť muže byt namapována do oblasti globální paměti. Přístupová doba bude menší než v případě konstantní paměti.

- **Privátní paměť (Private Memory)**

  Jedná se o paměť, která je unikátní pro každou instanci výpočetní jednotky neboli compute unit(work-item). Přístupová doba je nejmenší ze všech dostupných druhu paměti.



Obrázek 6: OpenCL paměťový model[zdroj: Khronos]

3.4.3 Optimalizace

Vzhledem k abstrakci celého OpenCL rozhraní je poměrně těžké získat nastavení, které by bylo automaticky nejlepší pro každé zařízení. Každé zařízeni má jiné slabiny. Ale některé skupiny zařízení mají občas stejné vlastnosti. Např pro grafické karty se jako úzké hrdlo jeví PCI-E sběrnice, proto se doporučuje minimalizovat objem přenesených dat, což by mělo pomoci výkonu. Pro GPU je nejdůležitější správné nastavení přístupu k paměti, kdy často platí čim méně se využívá globální paměť tím lepe.

3.4.4 Výhody a nevýhody

Jednou z hlavních výhod díky abstrakci je nezávislost na zařízení, ať už se jedná o grafické karty, kdy není problém provozovat OpenCL jak na grafických kartách AMD tak Nvidia. V budoucnu se k nim možná přidají i nějací další výrobci. Stejně tak je možné se správnou implementací provozovat OpenCL i na různých procesorech. Prakticky schopnost využívat OpenCL na zařízení závisí na tom jestli výrobce implementoval standard.

Bohužel abstrakce má i svou špatnou stránku, kdy pokud chceme dosáhnout lepšího výkonu na určitém zařízení musíme pro to zařízení optimalizovat, což nám ztíží možnost přenositelnosti na jiná zařízení, kde muže dojit k poklesu výkonu popřípadě to nemusí chodit.

Nvidia dále nabízí nástroje pro profilování kernelu, což umožňuje zjistit v čem by se dal kernel zrychlit. Tento nastroj se nazývá Nvidia Visual Profiler.

Další z výhod je nezávislost na operačním systému počítače, kdy pokud výrobce poskytuje ovladače, které implementují OpenCL tak nebude problém s jeho využitím.

Jako další výhoda se muže jevit použití c99 dialektu programovacího jazyka C, který už dlouhodobě patří mezi hodně využívané jazyky.

3.5 DirectCompute

Jedná se o API, které bylo vyvinuto firmou Microsoft jako součást DirectX. Bylo vyvinuto i s ohledem na využití grafických karet pro výpočty, ale hlavní využiti je spolupráce s DirectX na použití pro 3d aplikace. Toto API běží jen na operačních systémech firmy Microsoft a to konkretně Windows Vista, Windows 7 a novějších. Dále také jen na kartách, které podporují DirectX 10 a lepší. Celý výpočetní shader je založen na HLSL.

4 Návrh počítače pro výpočty pomocí GPU

4.1 Procesor

Už dlouho dobu se trh CPU určených do desktopu točí okolo 2 výrobců AMD a INTEL. V současné době se jeví v oblasti CPU lepší Intel, který vyniká zejména rychlostí na jedno jádro a spotřebou. Další parametry, které nás zajímají ohledně procesoru je počet jader, frekvence, velikost L2 a L3 cache a samozřejmě cena. Dále vzhledem k tomu že novější procesory jak od AMD tak INTEL mají integrovaný řadič paměti. Tak nás také zajímá maximální velikost a frekvence paměti které podporuje.

Došlo k výběru: *Intel Core i5-2310*

4.2 Základní deska

Výběr základní desky samozřejmě záleží na výběru procesoru, kdy se musí vybrat deska se stejnou patici jakou má procesor. Dále také záleží jestli má slot PCI-Express x16, který umožňuje připojení grafické karty. V současné době se nejčastěji využívá PCI-Express 2.0 , ale už existují i verze s 3.0, které nabízejí větší rychlost. A v oblasti výpočtu není rychlosti nikdy dost. Dále tu je podporovaná paměť a množství paměťových slotu. Formát

základní desky (micro ATX,ATX) je dalších parametrem, který následně i určuje velikost skříně. Další parametry už záleží na tom jestli potřebujeme nějaké specifické sloty popřípadě lepší chlazení a tak dále.

Došlo k výběru: *GIGABYTE GA-Z68P-DS3 - Intel Z68*

4.3 Grafická karta

Grafická karta je ve výběru nejdůležitější. Kdy nás zajímá celková rychlost čipu, zdali má podporu pro výpočty (v dnešní době podporují všechny novější karty). Dále jestli podporuje výpočty double precision a také jak rychle v nich umí počítat. Další věc kterou musíme rozhodnout je zda koupit speciální kartu na výpočty. U Nvidie se tyto karty nazývají Tesla popřípadě Quadro. U AMD je to FireStream, FireGL a další. Tyto karty se od obyčejných liší svou rychlosti, kdy např v double precision je rozdíl až desetinásobný. Zároveň s tím výrobce poskytuje speciální ovladače, které jsou testované a často i stabilnější než ty určené pro obyčejné karty. Bohužel tomuto rozdílu odpovídá i cena, která je několikanásobně vyšší než v případě obyčejných karet. Takže se vyplatí zauvažovat jestli se tento cenový rozdíl vyplatí zaplatit.

Problémem u obyčejných karet je právě výkon v double precision oproti single precision, kdy např u Nvidie karet se liší tento výkon podle řady např. Geforce 56x řada má výkon v double precision 1/12 single precision, u řady 57x a 58x je tento poměr zase 1/8. U profesionálních karet je tento poměr často okolo 1/2.

Došlo k výběru: GIGABYTE GTX 560 Ti Ultra Durable 1GB

4.4 Paměť

Vybírá se v závislosti na podporovaném druhu paměti u základní desky(slot) a procesoru. Rozlišujeme frekvenci,druh(SDR,DDR1,DDR2,...) a velikost. V současné době je velikost 4 GB považována na standardní. Často se ale vzhledem k cenně vyplatí zauvažovat o 8 GB

Došlo v výběru: *Kingston 4GB (kit 2x 2GB) 1333MHz*

4.5 Zdroj

Zdroj je v případě počítače, který má sloužit pro výpočty na grafické kartě velmi důležitá věc. Použitím výkonné grafické karty se požadavky na zdroj velmi zvětšují. Nároky se ještě vice zvětšují při využití více grafických karet. Zajímá nás celkový výkon, jeho účinnost a maximální proudový odběr na jednotlivých větvích. U grafických karet se často uvádí jaký výkon má mít zdroj aby tuto grafickou kartu utáhl.

Došlo k Výběru: *Corsair Builder Series CX600 V2 600W*

4.6 Harddisk

Pevný disk vybereme v závislosti na požadované kapacitě a rychlosti disku, která se určuje v závislosti na velikosti vyrovnávací paměti a rychlosti otáček. Dále se může lišit i úrovní hluku.

Došlo k Výběru: WD Caviar Blue WD5000AAKX 3.5" 500GB

4.7 Skříň

Velikost skříně závisí na vybrané základní desce, kdy se rozlišuje několik formátu (micro ATX,ATX,...). Dalším z mnoha úskalí, které nás muže potkat je samotná velikost grafické karty, kdy často platí že čím se jedná o výkonnější grafickou kartu tím je delší. Dále by měla by poskytovat dobré možnosti chlazení např možnost přidaní několika dalších větráčků popřípadě vyměnění chlazení CPU za lepší.

Došlo k výběru: Cooler Master Elite 370 černá

4.8 Mechanika

V současné době se nejčastěji kupují DVD-RW mechaniky, které se liší rychlostí vypalování podle media. Dále také podporou rozhraní, kdy se v dnešní době hodně využívá rozhraní SATA.

Došlo k výběru: Samsung SH-222AB, SATA, černá, Bulk

5 Příprava počítače

5.1 Nastavení Microsoft Windows 7

5.1.1 Instalace

Samotná instalace operačního systému Windows není zas taková obtížná záležitost. Po naběhnutí instalačního DVD systému Windows 7 nás uvíta obrazovka s výběrem jazyka,časové zóny a rozložení klávesnice. Na další obrazovce klikneme na nainstalovat. Dále dostaneme k výběru jakou verzi chceme nainstalovat, z hlediska použití tento výběr není zas takový důležitý, ale pokud chceme změnit jazyk OS po instalaci je nutné zvolit Ultimate verzi. Další věci kterou musíme zvolit je jestli chceme 32bit nebo 64 bit verzi Windows. To už se nás tyká, kdy největší výhoda 64 bit verze je podpora vice jak 3 GB paměti což se v našem případě hodí vzhledem k tomu že počítač obsahuje 4GB. Na další obrazovce si můžeme přečíst licenční podmínky. Po přijmutí licenčních podmínek nás dále čeká výběr jestli chceme upgradovat z nějaké stávající verze Windows popřípadě chceme čistou instalaci(zvolíme čistou). Poslední rozhodnutí, které nás čeká je na jaký disk a do jakého oddílu se má nainstalovat OS. Zvolíme požadovanou velikost oddílu. Pokud chceme nainstalovat Linux po instalaci Windows vyplatí se si nechat nějaké nepřidělené místo na disku do kterého se poté Linux může nainstalovat. Po kliknutí na další už začne samotná automatická instalace.

Po nainstalovaní se nám počítač restartuje a najede na obrazovku kde zadáme jméno počítače a osoby, která ho bude využívat. Jako další můžeme zadat heslo k učtu, který vytváříme. Poté zadáme sériový klič, který jsme dostali společně s DVD. Dále se objeví nastavení automatických aktualizací systému Windows, data a času a umístění počítače.

Poté dojde k automatickému dokončení nastavení a zobrazí se nám plocha.

5.1.2 Ovladače

Samotné ovladače můžeme najít buď na CD, které jsme dostali společně se zařízením. Nebo se dají stáhnout z webu výrobce což pokud nemáme k dispozici síť hned po instalaci můžeme provést z jiného počítače. Často se stává že ovladače na CD jsou poměrně staršího data proto se často doporučuje je stáhnout z internetu.

Instalace ovladačů by měla probíhat tak že nejdříve se nainstalují ovladače na základní desku a až poté se instalují ostatní periferie jako např grafická karta. V našem případě se nám zvuková a síťová karta nainstaluje společně s ovladači základní desky. Jelikož jsou obě integrované.

Nejdůležitější ovladačem je pro nás ovladač grafické karty, který nám po nainstalovaní zpřístupní možnost využit grafickou kartu pro výpočty. Lze použít jakýkoliv ovladač, ale nějaké jsou speciálně certifikované pro tuto práci. V současné době je to ovladač Nvidia 301.32 nebo 301.27 pro Windows

5.1.3 Visual Studio

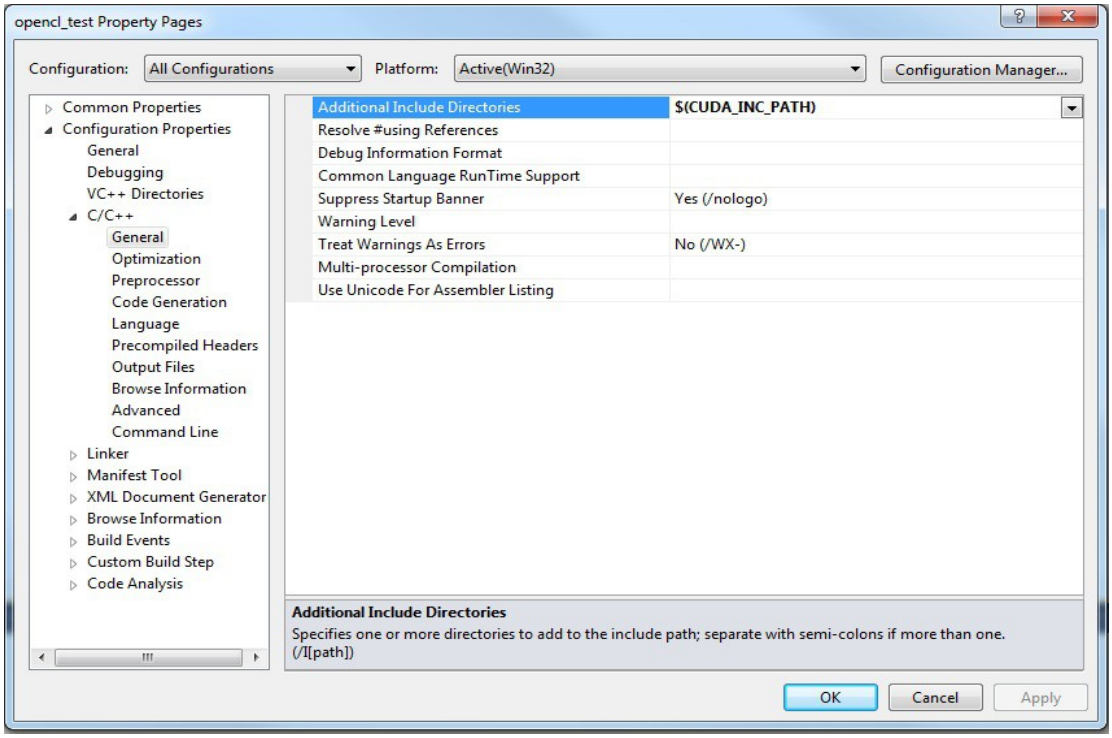
Visual studio si buď můžeme koupit popřípadě použít bezplatnou express verzi, kterou poté co se zaregistrujme na stránkách visual studia můžeme stáhnout. V případě OpenCL nám na verzi nezáleží..

5.1.4 Cuda Toolkit

Cuda Toolkit lze stáhnout ze stránek Nvidie, kdy máme na výběr mezi 32 bit nebo 64 bit verzí. Poté stačí akorát nainstalovat a zapamatovat si cestu kam se to nainstalovalo abychom to poté mohli využít ve Visual studiu.

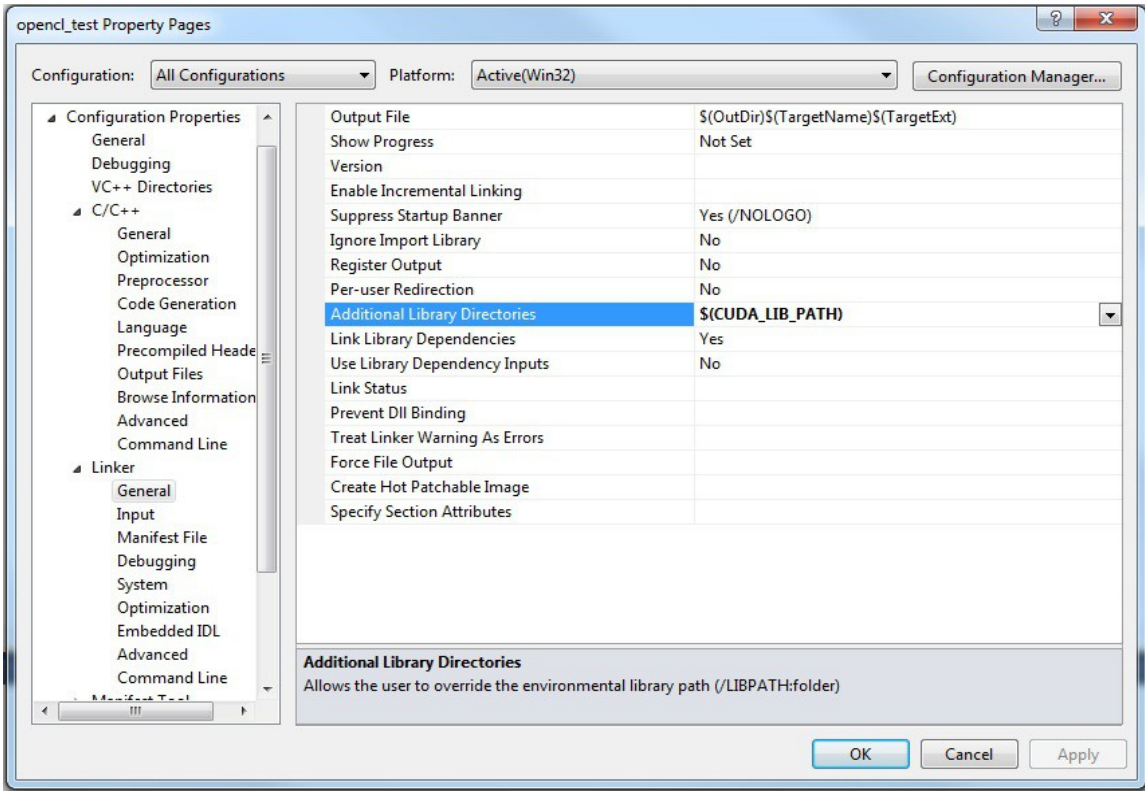
5.1.5 Nastavení projektu ve visual studio pro OpenCL

Vytvoříme nový prázdný c projekt a v něm vytvoříme soubor obsahující zdrojový kód. Poté klikneme na *Projects → Properties* a dostaneme se na konfigurační stránku projektu. První věci kterou musíme změnit je *Configuration*, kterou přepneme z *Active (Debug)* na „*All Configurations*“. Dále otevřeme *Configuration Properties → C/C++* , kde najedeme na *General* a přidáme do *Additional Include Directories* proměnou „*$ (CUDA_INC_PATH)*“. Která ukazuje kde se nachází include adresář nainstalovaného Cuda Toolkit.



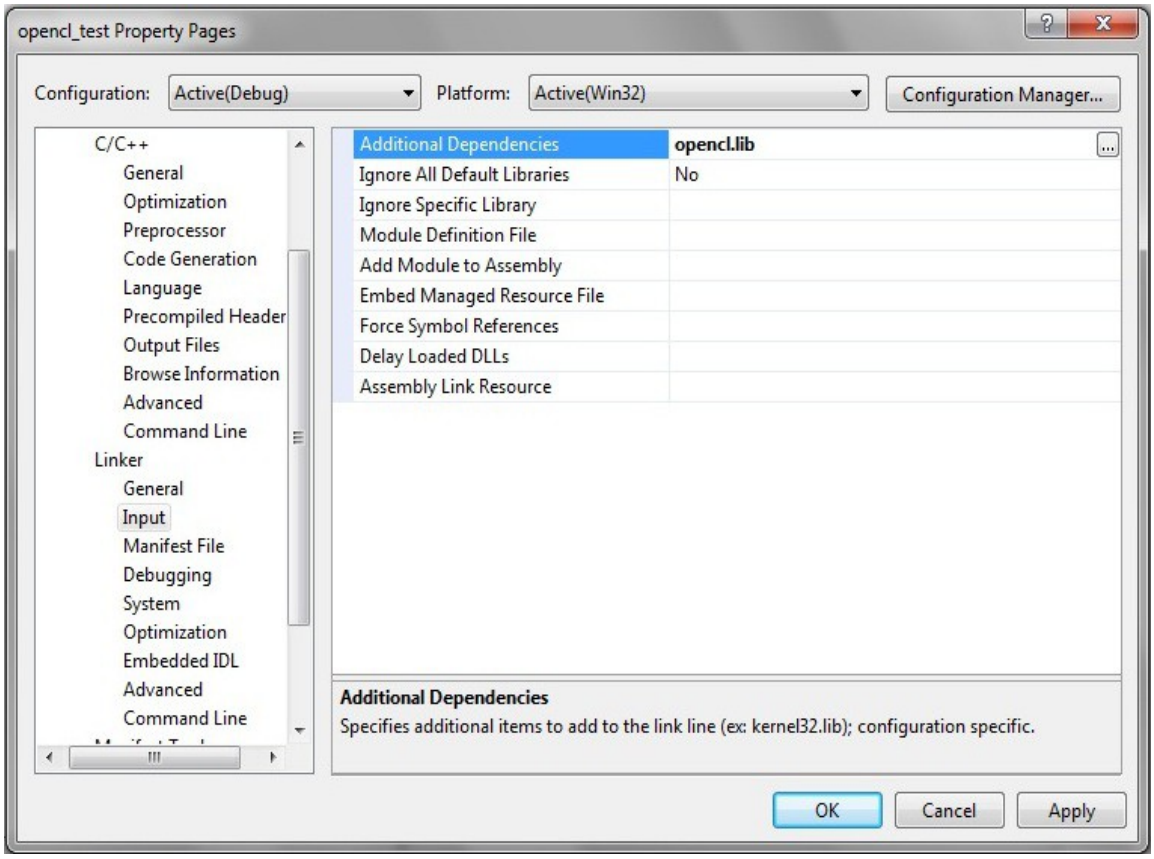
Obrázek 7: Include(Visual Studio)

Dále pokračujeme *Configuration Properties → Linker*, kde najedeme zase na General a snažíme se najít „*Additional Library Directories*“ do které zase napíšeme proměnou „*$(CUDA_LIB_PATH)*“ která ukazuje kde se nachází knihovny Cuda Toolkitu.



Obrázek 8: Nastavení knihovny(Visual Studio)

I když Visual Studio ví kde má hledat knihovny které potřebuje, musíme mu ukázat jakou knihovnu má přesně používat. Toto nastavení najdeme *Configuration Properties* → *Linker* v *Input*. Přesně se jedná o *Additional Dependencies* kde napíšeme „*OpenCL.lib*“.



Obrázek 9: Další závislosti(Visual Studio)

Po kliknutí na Apply by už měl být projekt nastaven pro OpenCL programování. Kompilace poté probíhá stejně jako u každého jiného projektu ve Visual Studiu.

5.2 Nastavení OpenSuse

5.2.1 Instalace

V dnešní době už i instalace linuxu není tak obtížnou věcí. Po vložení DVD nás uvítá menu, kde klikneme na instalaci. Jako první se ukáže licenční ujednání a volba jazyka. Vybereme jaký jazyk chceme a pokračujeme dál. Dále si vybereme jestli se jedná o novou instalaci popřípadě o upgrade stávající verze OpenSuse. V našem případě vybereme novou instalaci. Jako další máme nastavení časové zóny a samotného času a data. Po jejich nastavení nasleduje výběr desktopového prostředí. Po výběru desktopového prostředí následuje výběr na jaky oddíl se má opensuse nainstalovat. Vzhledem k tomu že jsme si při instalaci Windows nechali nevyužité místo, měl by nám instalátor sám nabídnout instalaci do

tohoto prostoru. Dále nás čeká vytvoření uživatele a jeho hesla, které zároveň slouží jako heslo pro použití sudo na elevaci práv. Po jeho vytvoření nás uvítá obrazovka obsahující všechna nastavení, která jsme provedli v rámci instalace. Pokud vše souhlasí tak můžeme pokračovat do dalšího kroku, kdy už začne samotná automatická instalace. Po jejím dokončení můžeme restartovat počítač a po výběru v GRUB menu by nám měl naběhnout operační systém OpenSuse.

5.2.2 Ovladače

Jsou 2 způsoby jak nainstalovat ovladače v Opensuse. A to buď instalace skrze repositář software. Nebo stáhnutí instalátoru ze stránek Nvidie, jeho označení jako spustitelný a následné spuštění. Poté akorát postupujeme podle instrukcí, které nám instalátor dává. Obecně je jednodušší instalace z repositáře, kdy dojde k automatické instalaci která je speciálně připravená pro distribuci na kterou ovladač instalujeme. Stejně jako v případě instalace ve Windows existují speciální verze ovladačů uzpůsobené pro výpočty, ale fungují dobře s většinou ovladačů.

V současné době jde OpenCL pouze na ovladačích od výrobce(Nvidia a AMD). V budoucnosti se počítá i s podporou u open source ovladačů.

5.2.3 Kompilátor

Jeden z hlavních kompilátoru pro Linux je bezpochyby GCC, který lze nainstalovat z repositářů distribuce. Tento kompilátor podporuje jak OpenCL tak zároveň i OpenMP, které se nám bude hodit pří porovnávání rychlosti s CPU.

5.2.4 Cuda Toolkit

Cuda Toolkit stáhneme ze stránek Nvidie. Dáme mu práva spouštění což uděláme pomocí příkazu *chmod +x jmeno*. Po spuštění tohoto souboru se chvíli bude rozbalovat. Po rozbalení dostaneme na výběr kam se má nainstalovat. Pokud ho chceme nainstalovat někam jinam než do home složky je potřeba ho spustit s právy roota. Cestu kam se Cuda toolkit nainstaloval si zapamatujeme. Bude to potřeba při kompilaci OpenCL kódu.

5.2.5 Kompilace projektů používajících OpenCL

Jako kompilátor pro tuto činnost budeme využívat GCC, které je k dispozici ve všech

distribucích. V tuhle chvíli kompilujeme pouze C zdroják, cl soubor obsahující kernel se nám automaticky zkompiluje při spuštění samotného programu. C soubor zkompilujeme pomocí tohoto příkazu.

```
gcc kod.c -o kod.x -L/usr/local/cuda/lib64 -I/usr/local/cuda/include -lOpenCL
```

Vzhledem k délce příkazu se vyplatí uvažovat o udělaní makefile, který kompilaci značně zjednodušuje. Ukázkový makefile je k dispozici v příloze.

6 Ukázka OpenCL na příkladu

6.1 Host soubor

```
cl_platform_id platform_id = NULL;
cl_device_id device_id = NULL;
cl_uint ret_num_devices;
cl_uint ret_num_platforms;
cl_int ret;
cl_event prof_event;
cl_context context;
cl_command_queue command_queue;
cl_program program;
cl_kernel kernel;
size_t global_item_size = velik_listu;
size_t local_item_size = 64;
```

Jedná se o základní proměnné OpenCL programu. Bude se s nimi pracovat v pozdější fázi programu, tam také dojde k vysvětlení co ukládají.

```
ret=clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
```

Cílem tohoto příkazu je získat počet platform, které jsou k dispozici.

Argumenty:

- První argument určuje počet platform, které se mají zapsat do platform_id.
- Druhý argument vrací počet platform, které byli nalezeny a ukládá je do proměnné platform_id.
- Třetí argument pote vrací počet platform, ale jako číslo a ukládá je do

ret_num_platforms.

```
ret=clGetDeviceIDs(platform_id,CL_DEVICE_TYPE_ALL,1,&device_id,
&ret_num_devices);
```

Tento příkaz má za úkol získat list zařízení, které jsou k dispozici na platformě.

Argumenty:

- Prvním argumentem je odkaz na platformu, kterou jsme získali pomocí příkazu clGetPlatformIDs
- Druhý argument určuje typ OpenCL zařízení, které se má využít. V našem případě se nám hodí buď CL_DEVICE_TYPE_ALL nebo CL_DEVICE_TYPE_GPU.
- Třetí argument určuje kolik zařízení se má přidat do proměnné device_id, která vrací list zařízení.
- Posledním argumentem je tzv ret_num_devices jež vrací počet zařízení číslem.

```
context=clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);
```

Tento příkaz má za úkol vytvoření contextu.

Argumenty:

- Prvním argumentem specifikuje list context nazvu a jejich proměnných.
- Druhý argument určuje počet zařízení, která jsou obsažená v device_id proměnné
- Třetí argument je ukazatel na proměnou s listem zařízení
- Čtvrtý argument funguje jako funkce pro vracení chyb, které mohou vzniknout v rámci tohoto contextu. Pokud je NULL nedojde k žádnému vracení chyb.
- Pátý argument bude zavolán v případě toho že bude zavolán čtvrtý argument. Jedná se o tzv user-data
- Posledním argumentem je vracení chybového kódu pokud dojde k chybě v tomto příkazu

```
command_queue = clCreateCommandQueue(context, device_id,
CL_QUEUE_PROFILING_ENABLE, &ret);
```

Command_queue slouží pro vytvoření fronty příkazů.

- Prvním argumentem je context
- Druhým argumentem je list zařízení, která jsou asociována s contextem. Jedná se o device_id list.
- Třetí argument určuje specifické vlastnosti pro command_queue. Např zdali má dojít ke spouštění příkazu postupně v určitém uspořádaní nebo víceméně náhodně. Další vlastností je jestli je povolené profilovaní příkazu. Což je v našem případě povolené abychom mohli zjistit za jakou dobu se vykoná kernel.
- Posledním argumentem je vracení chybového kódu pokud dojde k chybě v tomto příkazu

```
cl_mem cislo1_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
            velik_listu * sizeof(int), NULL, &ret);
cl_mem cislo2_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
            velik_listu * sizeof(int), NULL, &ret);
cl_mem cislo3_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
            velik_listu * sizeof(int), NULL, &ret);
cl_mem vysledek_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
            velik_listu * sizeof(int), NULL, &ret);
```

Cílem těchto příkazu je vytvoření bufferu. Buffer je jedno-dimenzionální pole.

Argumenty:

- Prvním argumentem je samotný context, který jsme už vytvořily příkazem clCreateContext
- Druhý argument se týká paměti. Kdy určujeme zda-li kernel bude moct zapisovat a číst proměnou(CL_MEM_READ_WRITE), popřípadě bude mocí jen číst(CL_MEM_READ_ONLY) a nebo jen zapisovat (CL_MEM_WRITE_ONLY).
- Třetí argument je velikost paměti, která se má alokovat

- Čtvrtý argument je ukazatelem na data, která už mohou byt alokována aplikací. Pokud je NULL řešíme alokaci pomocí clEnqueueWriteBuffer.
- Posledním argumentem je vracení chybového kódu pokud dojde k chybě v tomto příkazu

```
ret = clEnqueueWriteBuffer(command_queue, cislo1_mem_obj, CL_TRUE, 0,
            velik_listu * sizeof(int), cislo1, 0, NULL, NULL);
ret = clEnqueueWriteBuffer(command_queue, cislo2_mem_obj, CL_TRUE, 0,
            velik_listu * sizeof(int), cislo2, 0, NULL, NULL);
ret = clEnqueueWriteBuffer(command_queue, cislo3_mem_obj, CL_TRUE, 0,
            velik_listu * sizeof(int), cislo3, 0, NULL, NULL);
```

Tento příkaz má za úkol zapsat buffer z host paměti do paměti zařízení.

Argumenty:

- První argument odkazuje na command-queue, které bylo vytvořeno příkazem clCreateCommandQueue.
- Druhý argument poté odkazuje na samotný buffer, do kterého chceme něco nahrát.
- Třetí argument ukazuje zdali má tato operace blokovat popřípadě ne. Kdy v případě blokovaní program pokračuje dal až teprve tehdy, když se tato funkce dokončí. Blokování se určuje příkazem CL_TRUE. Pokud chceme aby nedocházelo k blokováni zvolíme CL_FALSE.
- Čtvrtý argument je tzv offset, kterým můžeme udělat aby došlo k zápisu až od určité hodnoty.
- Pátý argument je velikost dat, která se mají zapsat.
- Šestý je ukazatel na buffer v host paměti z kterého má dojít k zápisu.
- Sedmý argument specifikuje počet událostí, které je potřeba vykonat předtím než dojde ke spuštění tohoto příkazu. Pokud je tento argument NULL musí event_wait_list byt také NULL.
- Osmí argument je event_wait_list, který specifikuje jaké události je potřeba vykonat předtím než dojde ke spuštění tohoto příkazu.

- Posledním argumentem je objekt, který bude odkazovat přesně na tuto funkci.

```
program = clCreateProgramWithSource(context,1,(const char **)&zdroj_kod,
NULL, &ret);
```

Cílem toho příkazu je nahrát zdrojový kód kernelu.

Argumenty:

- Prvním argumentem je odkaz na context vytvořený příkazem clCreateContext.
- Druhým argumentem je count odkaz.
- Třetím argumentem je odkaz na pole, které obsahuje zdrojový kód kernelu.
- Čtvrtým argumentem je pole s počtem znaků v každém stringu. Pokud je NULL jsou všechny stringy NULL-terminated.
- Posledním argumentem je vracení chybového kódu pokud dojde k chybě v tomto příkazu.

```
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
```

Zkompiluje a zlinkuje zdrojový kód načtený pomocí clCreateProgramWithSource

Argumenty:

- Prvním argumentem je odkaz na objekt. Který byl vytvořen pomocí clCreateProgramWithSource.
- Druhým argumentem je počet zařízení obsažených v proměnné device_id.
- Třetím argumentem je odkaz na list zařízení device_id
- Čtvrtý argument určuje vlastnosti kompilace.
- Pátým argumentem je notifikace, která se zavolá po úspěšné popřípadě neúspěšné kompilaci.
- Poslední argument bude zavolán v případě toho že bude zavolán pátý argument. Jedná se o tzv user-data

```
kernel = clCreateKernel(program, "scitani_rov", &ret);
```

Vytvoří kernel objekt a určí funkci, která se má použit v kernelu v našem případě scitani_rov.

Argumenty:

- Prvním argumentem je odkaz na objekt. Který byl vytvořen pomocí clCreateProgramWithSource
- Druhým argumentem je název funkce, která se má spustit a kterou najdeme v kernel kódu.
- Posledním argumentem je vracení chybového kódu pokud dojde k chybě v tomto příkazu

```
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), void *)&cislo1_mem_obj);
ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), void *)&cislo2_mem_obj);
ret = clSetKernelArg(kernel, 2, sizeof(cl_mem),(void *)&cislo3_mem_obj);
ret = clSetKernelArg(kernel, 3, sizeof(cl_mem),(void *)&vysledek_mem_obj);
```

Jedná se o nastavení kernel argumentu

Argumenty:

- Prvním argumentem je tzv kernel objekt který jsme vytvořily pomocí clCreateKernel příkazu.
- Druhým argumentem se určuje pořadí, které poté musíme dodržet při psaní argumentu kernelu. Samotné názvy proměnných se v rámci kernelu mohou lišit.
- Třetí argument určuje velikost kernel argumentu, který chceme použít.
- Poslední argument odkazuje na samotná data, která se mají použít v rámci kernelu.

```
ret=clEnqueueNDRangeKernel(command_queue,kernel,1,NULL,&global_item_size,
&local_item_size, 0, NULL, &prof_event);
```

Příkaz pro spuštění kernelu na zařízení.

Argumenty:

- Prvním argumentem je odkaz na command_queue.
- Druhým argumentem je odkaz na kernel objekt.

- Třetím argumentem je určení dimenze s kterou budou pracovat globální work-items a work-items ve skupině. Musí být větší než 0.
- Čtvrtým argumentem je offset, který se má započítat do vypočtení global ID. Pokud NULL global IDs začínají na (0,0,...).
- Pátým argumentem je global_work_size, který určuje počet work-items v určené dimenzi. Např v případě že máme 2 listy s 250 hodnotami a chceme je sečíst. Tak v tomto případě bude global_work_size=250.
- Šestým argumentem je local_work_size. Určuje kolik work-items je ve skupině(work-group). A musí být vždy menší než global_work_size
- Sedmý argument specifikuje počet událostí, které je potřeba vykonat předtím než dojde k vykonání tohoto příkazu. Pokud je tento argument NULL musí event_wait_list byt také NULL.
- Osmý argument je event_wait_list, který specifikuje jaké události je potřeba vykonat předtím než dojde k vykonaní tohoto příkazu.
- Posledním argumentem je objekt, který bude odkazovat přesně na tuto funkci. V tomto případě toho využíváme abychom zjistili jak dlouho trvala kernelu exekuce.

```
ret = clWaitForEvents(1, &prof_event);
    clFinish(command_queue);
    clGetEventProfilingInfo(prof_event, CL_PROFILING_COMMAND_START,
            sizeof(time_start), &time_start, NULL);
    clGetEventProfilingInfo(prof_event, CL_PROFILING_COMMAND_END,
            sizeof(time_konec), &time_konec, NULL);
    total_time = (time_konec - time_start);
```

Tento úsek kódu slouží k získaní času po která kernel běžel. Výsledek je v nanosekundách.

```
ret = clEnqueueReadBuffer(command_queue, vysledek_mem_obj, CL_TRUE, 0,
velik_listu * sizeof(int), vysledek, 0, NULL, NULL);
```

Tento příkaz má za úkol převést buffer z paměti zařízení do host paměti.

Argumenty:

- První argument odkazuje na command-queue, které bylo vytvořeno příkazem clCreateCommandQueue.
- Druhý argument pote odkazuje na samotný buffer z kterého chceme něco přečíst. Tento buffer se vyskytuje na zařízení.
- Třetí argument ukazuje zdali má tato operace blokovat popřípadě ne. Kdy v případě blokovaní program pokračuje dal až teprve tehdy když se tato funkce dokončí. Blokování se určuje příkazem CL_TRUE. Pokud chceme aby nedocházelo k blokováni zvolíme CL_FALSE.
- Čtvrtý argument je tzv offset kterým můžeme udělat aby došlo k zápisu až od určité hodnoty.
- Pátý argument je velikost dat, která se mají přečíst.
- Šestý je ukazatel na buffer v host paměti do kterého se má zapsat..
- Sedmí argument specifikuje počet událostí, které je potřeba vykonat předtím než dojde k vykonání tohoto příkazu. Pokud je tento argument NULL musí event_wait_list byt také NULL.
- Osmí argument je event_wait_list, který specifikuje jaké události je potřeba vykonat předtím než dojde k vykonaní tohoto příkazu.
- Posledním argumentem je objekt, který bude odkazovat přesně na tuto funkci.

```
ret = clFlush(command_queue);
ret = clFinish(command_queue);
ret = clReleaseKernel(kernel);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(cislo1_mem_obj);
ret = clReleaseMemObject(cislo2_mem_obj);
ret = clReleaseMemObject(cislo3_mem_obj);
ret = clReleaseMemObject(vysledek_mem_obj);
ret = clReleaseCommandQueue(command_queue);
ret = clReleaseContext(context);
```

Po dokončení výpočtu po sobě uklidíme.

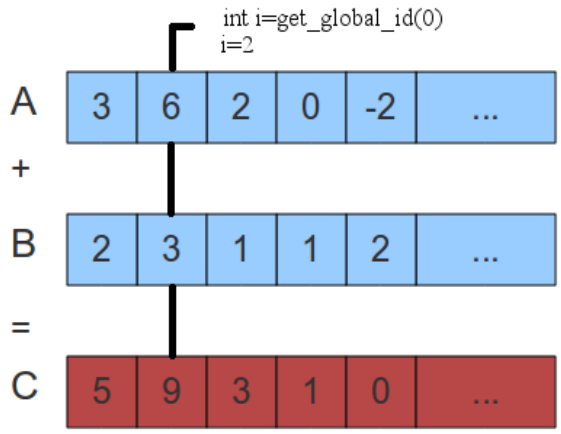
6.2 Kernel

```
__kernel void scitani_rov(__global int *cislo1, __global int *cislo2,
__global int *vysledek) {
   // ziskat index
    int i = get_global_id(0);
    // secist
    vysledek[i] = cislo1[i] + cislo2[i];
    }
```

Samotná funkce kernelu, která se bude spouštět na zařízení musí byt označena __kernel předponou. V závorce se potom nachází samotné argumenty kernelu, které musí dodržovat pořadí, které bylo zadáno pomocí příkazu setKernelArg v kódu hosta. Samotný název proměnných, ale může být rozdílný. Toto je ukázkový kernel pro vypočtení rovnice C=A+B pro velké množství hodnot. Samotný kernel by se dal přirovnat k tomuto příkladu z C.

```
for(i = 0; i < velik_listu; i++) {
    C[i] = A[i] + B[i];
    }
```

Kdy get_global_id nám označuje jednotlivé work-items s jejich daty a zajišťuje nám tak aby každý work-item sčítal jiná data. Místo velik_listu máme global_work_size v host souboru.



Obrázek 10: Sčítání-ukazka[zdroj:theBigBlob]

6.3 Další zajímavé příkazy

Pokud potřebujeme získat informace o zařízení můžeme využít clGetDeviceInfo. Příkaz clCreateImage2D slouží pro vytvoření obrázkového objektu, který se využívá pro skladování jedno,dvou či tří-dimenzionální textury,frame-buffery či obrázku. Pokud máme zdrojový kód kernelu ve formě binárního souboru používá pro načtení clCreateProgramWithBinary.

7 Porovnání rychlosti

7.1 Specifikace počítače

Počítač číslo 1:

- procesor: Intel Core i5-2310 box 2,9 Ghz
- základní deska: Gigabyte GA-Z68P-DS3
- grafická karta: Gigabyte GTX 560 Ti Ultra Durable 1GB
- operační paměť: Kingston 4GB
- Operační systém: Opensuse 12.1 64bit

Počítač číslo 2:

- procesor: AMD Athlon II x3 435 2,9 Ghz
- základní deska: Gigabyte GA-MA770T-UD3
- grafická karta: MSI N240GT GeForce GT 240 512 MB GDDR5
- operační paměť: A-Data 4GB
- Operační systém: Arch Linux 32 bit

Měření probíhalo pro každý čas třikrát. Výsledkem je poté průměr naměřených hodnot. Jak OpenCL tak standardní C soubor(i s openMP) počítají jen dobu provedení funkce(kernelu). Není do toho započítána alokace pamětí a nahrávání dat popřípadě čtení dat ze zařízení.

7.2 Násobení matic

Výsledek počítače číslo 1.

Násobení Matic			
matice	CPU[s]	CPU(openmp)[s]	GPU[s]
512	0,1904	0,0564	0,0029
1024	9,0270	2,4098	0,0276
2048	58,1939	4,1778	0,2231
4096	552,9544	41,1230	1,9557

Výsledek počítače číslo 2.

Násobení Matic 2			
matice	CPU[s]	CPU(openmp)[s]	GPU[s]
512	1,2070	0,4262	0,0182
1024	11,4847	3,8221	0,1494
2048	108,8380	36,3984	1,1851
4096	567,8914	136,2196	-

U počítače číslo 2 došlo v průběhu násobení matic 4096x4096 pomocí GPU k zásek. Proto není uveden žádný výsledek.

Už i při nejmenší matici se ukazuje výkon ve prospěch gpu, tento náskok se se zvětšující se maticí zvyšuje. Při 4096 matici je už rozdíl ve výkonu téměř dvacetinásobný.

7.5 Transponování matice

Výsledky počítače číslo 1:

Transponování matice			
matice	CPU[ms]	CPU(openmp)[ms]	GPU[ms]
512	1,324	0,416	0,136
1024	8,603	2,595	1,139
2048	42,762	14,157	6,279
4096	215,761	63,535	26,311

Výsledky počítače číslo 2:

Transponování matice 2			
matice	CPU[ms]	CPU(openmp)[ms]	GPU[ms]
512	2,626	3,514	0,937
1024	14,909	8,518	3,800
2048	70,471	33,620	24,315
4096	667,612	294,272	99,915

I když menší než v případě násobení matic i tady je nějaký rozdíl vidět.

8 Závěr

Možností programování GPU jsou v současnosti mnohem lepší a jednoduší než byli kdy v minulosti. V současné době se uvažuje hlavně o 2 standardech. A těmi jsou Cuda C a OpenCL. Oba mají své výhody a nevýhody.

Z těchto dvou nakonec došlo k vybraní OpenCL, které si mě získalo svou nezávislostí na výrobci grafické karty a multiplatformností. Zároveň se zda že Cuda C je pomalinku na ústupu a právě OpenCL se dobývá na jeho pozici.

Samotná instalace podpory OpenCL není nic těžkého. Prakticky je potřeba akorát stáhnout SDK od výrobce,nainstalovat ho a následně nastavit vývojové prostředí. Samozřejmostí je také instalace kompilátoru. I výrobci se snaží celou věc značně ulehčovat a nabízejí spoustu aplikací pro usnadnění.

Podle naměřených výsledků poskytuje GPU celkem zajímavý nárůst výkonu v určitých operacích. Nejlépe se mu daří u funkci, které mají provést nějakou operací na velkém množství dat.

Seznam použité literatury

[1] D. Triolet, "Nvidia CUDA: preview," BeHardware, 2007 [cit. 2012-01-26]. Dostupné z:: http://www.behardware.com/articles/659-1/nvidia-cuda-preview.html.

[2]*CUDA programming*[online]. 2011 [cit. 2012-01-26]. Dostupné z: http://reference.wolfram.com/mathematica/CUDALink/tutorial/Programming.zh.html

[3]OpenCL Specification 1.1[online]. 2011 [cit. 2012-01-14] Dostupné z: http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[4] SANDERS, Jason. *CUDA by example*: *an introduction to general-purpose GPU programming*. 1st print. Upper Saddle River: Addison-Wesley, 2010, 290 s. ISBN 978-0-13-138768-3.

[5]OpenCL Work Item Ids: Global/Group/Local[online] 2012 [cit. 2012-04-19]. Dostupné z: http://jorudolph.wordpress.com/2012/02/03/opencl-work-item-ids-globalgrouplocal/

[6]OpenCL Programming Guide[online] 2010 [cit. 2012-01-15] Dostupné z: http://developer.nvidia.com/opencl

[7]Linking and Compiling OpenCL 2010 [cit. 2012-01-15] Dostupné z: http://www.guineacode.com/2010/linking-and-compiling-opencl/

[8]General-Purpuse Computation on Graphics Hardware[online] 2011 [cit. 2011-12-26] Dostupné z: http://gpgpu.org/

[9] NVIDIA: Cuda zone [online]. 2011 [cit. 2011-12-26] Dostupné z: http://developer.nvidia.com/opencl

[10]OpenCL[online]. 2011 [cit. 2012-01-15] Dostupné z: http://www.khronos.org/opencl/

[11]OpenCL Specification 1.1[online]. 2011 [cit. 2012-01-14] Dostupné z: http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[12] NVIDIA, "NVIDIA OpenCL - Best Practices Guide," [Online]. 2012 [cit. 2012-02-26] Dostupné z: http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf

Seznam příloh

A Zdrojové kódy

A.1 OpenCL násobení matic

A.1.1 Makefile

A.1.2 Kernel

A.1.3 Host soubor

A.2 Násobení matic v C

A.3 OpenCL Transponování matice

A.3.1 MakeFile

A.3.2 Kernel

A.3.3 Host soubor

A.4 Transponování matice v C

B Obsah CD

A Zdrojové kódy

A.1 OpenCL – Násobení Matic

A.1.1 Makefile

```
PROJ=nasobmatic
CC=gcc
CFLAGS=-std=c99 -Wall -DUNIX -g -DDEBUG -O3
CUDA_INSTALL_PATH =/usr/local/cuda
CUDA=${CUDA_INSTALL_PATH}
# 64-bit nebo 32-bit
PROC_TYPE = $(strip $(shell uname -m | grep 64))
LIBS=-lOpenCL
ifeq ($(PROC_TYPE),)
      CFLAGS+=-m32
else
      CFLAGS+=-m64
endif
# koukni se po AMD
ifdef AMDAPPSDKROOT
   INC_DIRS=. $(AMDAPPSDKROOT)/include
      ifeq ($(PROC_TYPE),)
            LIB_DIRS=$(AMDAPPSDKROOT)/lib/x86
      else
            LIB_DIRS=$(AMDAPPSDKROOT)/lib/x86_64
      endif
else
# Koukni se po nvidia
ifdef CUDA
   INC_DIRS=. $(CUDA)/include
      ifeq ($(PROC_TYPE),)
            LIB_DIRS=$(CUDA)/lib32
      else
            LIB_DIRS=$(CUDA)/lib64
      endif
endif
endif
$(PROJ): $(PROJ).c
```

```
	$(CC) $(CFLAGS) -o $@ $^ $(INC_DIRS:%=-I%) $(LIB_DIRS:%=-L%) $(LIBS)
.PHONY: clean
clean:
	rm $(PROJ)
```

A.1.2 Kernel

```
__kernel void nasobenimatic(__global float *mA,
                            __global float *mB,
                            __global float *mO,
                            uint widthA, uint widthB)
{
  int globalIdx = get_global_id(0);
  int globalIdy = get_global_id(1);
  float sum =0;
  for (int i=0; i< widthA; i++)
  {
        float tempA = mA[globalIdy * widthA + i];
        float tempB = mB[i * widthB + globalIdx];
        sum += tempA * tempB;
  }
  mO[globalIdy * widthA + globalIdx] = sum;
}
```

A.1.3 Host file

```
#include <stdio.h>
#include <stdlib.h>
#include "CL/cl.h"

#define RADKY 2048
#define SLOUPCE 2048

int main ()
{
  cl_float *matice1;
  cl_float *matice2;
  cl_float *vysledek;
  cl_uint sirka = SLOUPCE;
  cl_uint vyska = RADKY;
```

```
cl_ulong time_start, time_konec;
long total_time;

FILE *fp;
char *zdroj_kod;
long delka_souboru;
long delka_cteni;

fp = fopen("nasobmatic.cl","r");
    fseek(fp,0L, SEEK_END);
    delka_souboru = ftell(fp);
    rewind(fp);

    zdroj_kod = malloc(sizeof(char)*(delka_souboru+1));
    delka_cteni = fread(zdroj_kod,1,delka_souboru,fp);
    if(delka_cteni!= delka_souboru)
    {
       printf("Nelze přečist soubor\n");
       exit(1);
    }

zdroj_kod[delka_souboru+1]='\0';


int x,y;
matice1 = malloc(sizeof(cl_float)*sirka*vyska);
matice2 = malloc(sizeof(cl_float)*sirka*vyska);
vysledek = malloc(sizeof(cl_float)*sirka*vyska);

    for(y=0;y<vyska;y++)
    {
          for(x=0;x<sirka;x++)
          {
             matice1[y*vyska+x]= rand() % 6 + 1; //data
             matice2[y*vyska+x]= rand() % 6 + 1;
             vysledek[y*vyska+x]=0;

          }
```

```
    }

  //opencl promenne
  cl_platform_id platform_id = NULL;
    cl_device_id device_id = NULL;
    cl_uint ret_num_devices;
    cl_uint ret_num_platforms;
    cl_event prof_event;

  cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
  ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_ALL, 1,
            &device_id, &ret_num_devices);

    // vytvorime OpenCL context
   cl_context context = clCreateContext( NULL, 1, &device_id, NULL, NULL,
&ret);

    // vytvorime command queue
      cl_command_queue   command_queue   =   clCreateCommandQueue(context,
device_id, CL_QUEUE_PROFILING_ENABLE, &ret);

   // nahrajeme zdrojový kod opencl kernelu
  cl_program program = clCreateProgramWithSource(context, 1,
        (const char **)&zdroj_kod,NULL, &ret);

  // vytvoříme program
  ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

    // zkompilujeme opencl kernel
  cl_kernel kernel = clCreateKernel(program, "nasobenimatic", &ret);

  // Vytvoříme buffer objekty a nahrajeme matice do nich
     cl_mem   vstupni_buffer1   =   clCreateBuffer(context,CL_MEM_READ_ONLY|
CL_MEM_COPY_HOST_PTR, sizeof(cl_float) * RADKY*SLOUPCE, matice1, NULL);
     cl_mem   vstupni_buffer2   =   clCreateBuffer(context,CL_MEM_READ_ONLY|
CL_MEM_COPY_HOST_PTR, sizeof(cl_float) * RADKY*SLOUPCE, matice2, NULL);
    cl_mem  vystupni_buffer  =  clCreateBuffer(context,  CL_MEM_WRITE_ONLY,
sizeof(cl_float) * RADKY*SLOUPCE, NULL ,NULL);

  //nastavime kernel argumenty
```

```
  ret = clSetKernelArg(kernel, 0, sizeof(cl_mem),&vstupni_buffer1);
  ret = clSetKernelArg(kernel, 1, sizeof(cl_mem),&vstupni_buffer2);
  ret = clSetKernelArg(kernel, 2, sizeof(cl_mem),&vystupni_buffer);
  ret = clSetKernelArg(kernel, 3, sizeof(cl_uint),&sirka);
  ret = clSetKernelArg(kernel, 4, sizeof(cl_uint),&sirka);

  // nastavíme global work size
  size_t global[2];

  global[0]= sirka;
  global[1]= vyska;

  ret = clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL,
           global, NULL, 0, NULL, &prof_event);

  ret = clWaitForEvents(1, &prof_event);
    clFinish(command_queue);
    clGetEventProfilingInfo(prof_event, CL_PROFILING_COMMAND_START,
           sizeof(time_start), &time_start, NULL);
    clGetEventProfilingInfo(prof_event, CL_PROFILING_COMMAND_END,
           sizeof(time_konec), &time_konec, NULL);
    total_time = (time_konec - time_start);

     ret = clEnqueueReadBuffer(command_queue, vystupni_buffer, CL_TRUE, 0,
sizeof(cl_float)*sirka*vyska, vysledek, 0, NULL, NULL);

    float timeTaken=(float)total_time*1e-9;
   printf("Time taken = %.4lf seconds\n", timeTaken);

  clReleaseMemObject(vstupni_buffer1);
  clReleaseMemObject(vstupni_buffer2);
  clReleaseMemObject(vystupni_buffer);
  clReleaseProgram(program);
  clReleaseKernel(kernel);
  clReleaseCommandQueue(command_queue);
  clReleaseContext(context);
}
```

A.2 Násobení matic v C

```
/*
 * nasobeni matic pomoci openMP a bez
*/
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>
#include <assert.h>
#define MATICE 4096

int main(int argc, char **argv)
{
    int i,j,k;
    int n;
    int temp;
    double time_start, time_end;

    n=MATICE;
    int **mat1 = malloc( sizeof(int*) * n);
    int **mat2 = malloc( sizeof(int*) * n);
    int **vysledMatic = malloc( sizeof(int*) * n);

    for(i=0; i<n; ++i) {
        mat1[i] = malloc( sizeof(int) * n );
        mat2[i] = malloc( sizeof(int) * n );
        vysledMatic[i] = malloc( sizeof(int) * n );
    }

    srand( time(NULL) );

    for(i=0; i<n; ++i) {
        for(j=0; j<n; ++j) {
            mat1[i][j] = (rand() % n);
            mat2[i][j] = (rand() % n);
        }
    }
```

```
    printf("Bez OpenMP...");
    fflush(stdout);
    time_start = omp_get_wtime();

    for(i=0; i<n; ++i) {
        for(j=0; j<n; ++j) {
            temp = 0;
            for(k=0; k<n; ++k) {
                temp += mat1[i][k] * mat2[k][j];
            }
            vysledMatic[i][j] = temp;
        }
    }

    time_end = omp_get_wtime();
    printf(" %f sekund.\n", time_end-time_start);
    printf("Pomoci OpenMP..");
    fflush(stdout);


time_start = omp_get_wtime();
#pragma omp parallel for private(i, j, k, temp)
    for(i=0; i<n; ++i) {
        for(j=0; j<n; ++j) {
            temp = 0;
            for(k=0; k<n; ++k) {
                temp += mat1[i][k] * mat2[k][j];
            }
            vysledMatic[i][j] = temp;
        }
    }


    time_end = omp_get_wtime();
    printf(" %f sekund.\n", time_end-time_start);
    return 0;
}
```

A.3 OpenCL – Transponování matic

A.3.1 Makefile

```
PROJ=transpon
CC=gcc
CFLAGS=-std=c99 -Wall -DUNIX -g -DDEBUG
CUDA_INSTALL_PATH =/usr/local/cuda
CUDA=${CUDA_INSTALL_PATH}
# 64-bit nebo 32-bit
PROC_TYPE = $(strip $(shell uname -m | grep 64))
LIBS=-lOpenCL
ifeq ($(PROC_TYPE),)
      CFLAGS+=-m32
else
      CFLAGS+=-m64
endif
# koukni se po AMD
ifdef AMDAPPSDKROOT
   INC_DIRS=. $(AMDAPPSDKROOT)/include
      ifeq ($(PROC_TYPE),)
            LIB_DIRS=$(AMDAPPSDKROOT)/lib/x86
      else
            LIB_DIRS=$(AMDAPPSDKROOT)/lib/x86_64
      endif
else
# Koukni se po nvidia
ifdef CUDA
   INC_DIRS=. $(CUDA)/include

      ifeq ($(PROC_TYPE),)
            LIB_DIRS=$(CUDA)/lib32
      else
            LIB_DIRS=$(CUDA)/lib64
      endif
endif
endif
```

```
$(PROJ): $(PROJ).c
	$(CC) $(CFLAGS) -o $@ $^ $(INC_DIRS:%=-I%) $(LIB_DIRS:%=-L%) $(LIBS)
.PHONY: clean
clean:
	rm $(PROJ)
```

A.3.2 Kernel

```
__kernel void transponovani(__global float * puvodMatice,__global float * vysledek, const    uint    sirka)
{
	uint x = get_global_id(0);
	uint y = get_global_id(1);
	uint vysledIndex  = x*sirka     + y;
	uint zdrojIndex  = y*sirka       + x;
	vysledek[vysledIndex] = puvodMatice[zdrojIndex];
}
```

A.3.3 Host soubor

```
#include <stdio.h>
#include <stdlib.h>
#include "CL/cl.h"

#define RADKY 4096
#define SLOUPCE 4096

int main ()
{
  cl_float *puvodniMatice;
  cl_float *vysledek;
  cl_uint sirka = SLOUPCE;
  cl_uint vyska = RADKY;
  cl_ulong time_start, time_konec;
  long total_time;

  FILE *fp;
  char *zdroj_kod;
  long delka_souboru;
  long delka_cteni;
```

```
fp = fopen("transpon.cl","r");
    fseek(fp,0L, SEEK_END);
    delka_souboru = ftell(fp);
    rewind(fp);

    zdroj_kod = malloc(sizeof(char)*(delka_souboru+1));
    delka_cteni = fread(zdroj_kod,1,delka_souboru,fp);
    if(delka_cteni!= delka_souboru)
    {
       printf("Nelze přečist soubor\n");
       exit(1);
    }

zdroj_kod[delka_souboru+1]='\0';

int x,y;
int data=0;
puvodniMatice = malloc(sizeof(cl_float)*sirka*vyska);
vysledek = malloc(sizeof(cl_float)*sirka*vyska);
    for(y=0;y<vyska;y++)
    {
          for(x=0;x<sirka;x++)
          {
             puvodniMatice[y*vyska+x]= rand() % 6 + 1; //data
             vysledek[y*vyska+x]=0;
                  data++;

          }
    }

//opencl promenne
cl_platform_id platform_id = NULL;
  cl_device_id device_id = NULL;
  cl_uint ret_num_devices;
  cl_uint ret_num_platforms;
  cl_event prof_event;

cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
```

```
  ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_ALL, 1,
            &device_id, &ret_num_devices);

    // vytvorime OpenCL context
   cl_context context = clCreateContext( NULL, 1, &device_id, NULL, NULL,
&ret);

    // vytvorime command queue
      cl_command_queue   command_queue   =   clCreateCommandQueue(context,
device_id, CL_QUEUE_PROFILING_ENABLE, &ret);

   // Nacteme zdrojovy kod opencl kernelu
  cl_program program = clCreateProgramWithSource(context, 1,
        (const char **)&zdroj_kod,NULL, &ret);

  // Build the program
  ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

    // zkompilujeme opencl kernel
  cl_kernel kernel = clCreateKernel(program, "transponovani", &ret);

  // vytvoř buffery na zařízení a zkopiruj tam puvodni matici
     cl_mem   vstupni_buffer1   =   clCreateBuffer(context,CL_MEM_READ_ONLY|
CL_MEM_COPY_HOST_PTR,  sizeof(cl_float)  *  RADKY*SLOUPCE,  puvodniMatice,
NULL);
    cl_mem  vystupni_buffer  =  clCreateBuffer(context,  CL_MEM_WRITE_ONLY,
sizeof(cl_float) * RADKY*SLOUPCE, NULL ,NULL);

  //nastavime kernel argumenty
  ret = clSetKernelArg(kernel, 0, sizeof(cl_mem),&vstupni_buffer1);
  ret = clSetKernelArg(kernel, 1, sizeof(cl_mem),&vystupni_buffer);
  ret = clSetKernelArg(kernel, 2, sizeof(cl_uint),&sirka);

  // nastavíme globalní a localní work size
  size_t global[2];

  global[0]= sirka;
  global[1]= vyska;

  ret = clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL,
```

```
            global, NULL, 0, NULL, &prof_event);

  ret = clWaitForEvents(1, &prof_event);
    clFinish(command_queue);
    clGetEventProfilingInfo(prof_event, CL_PROFILING_COMMAND_START,
            sizeof(time_start), &time_start, NULL);
    clGetEventProfilingInfo(prof_event, CL_PROFILING_COMMAND_END,
            sizeof(time_konec), &time_konec, NULL);
    total_time = (time_konec - time_start);

     ret = clEnqueueReadBuffer(command_queue, vystupni_buffer, CL_TRUE, 0,
sizeof(cl_float)*sirka*vyska, vysledek, 0, NULL, NULL);

    float timeTaken=(float)total_time*1e-6;
   printf("Time Taken = %.4lf ms\n", timeTaken);
 /*
   printf("\nPuvodni matice \n");
      for(y=0;y<vyska;y++)
      {
       for(x=0;x<sirka;x++)
         {
            printf("%.2f , ",puvodniMatice[y*vyska+x]);
         }
         printf("\n");
      }
   printf("\nTransponovana matice \n");
      for(y=0;y<vyska;y++)
      {
       for(x=0;x<sirka;x++)
         {
            printf("%.2f , ",vysledek[y*vyska+x]);
         }
         printf("\n");
      }
   */

  clReleaseMemObject(vstupni_buffer1);
  clReleaseMemObject(vystupni_buffer);
```

```
  clReleaseProgram(program);
  clReleaseKernel(kernel);
  clReleaseCommandQueue(command_queue);
  clReleaseContext(context);
}
```

A.4 Transponování matice v C

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>
#include <assert.h>
#define MATICE 4096

int main()
{

  int i,j;
    int n;
    double time_start, time_end;
    n=MATICE;

    int **mat = malloc( sizeof(int*) * n);
    int **vysledMatic = malloc( sizeof(int*) * n);

    for(i=0; i<n; ++i) {
        mat[i] = malloc( sizeof(int) * n );
        vysledMatic[i] = malloc( sizeof(int) * n );
    }

    srand( time(NULL) );

    for(i=0; i<n; ++i) {
        for(j=0; j<n; ++j) {
            mat[i][j] = (rand() % n);
        }
    }
```

```
printf("Bez OpenMP...");
time_start = omp_get_wtime();
for(i=0;i<n;i++)
{
  for(j=0;j<n;j++) {
    vysledMatic[i][j]=mat[j][i];
  }
}

time_end = omp_get_wtime();
printf(" %f sekund.\n", time_end-time_start);

printf("Pomoci OpenMP..");
fflush(stdout);
time_start = omp_get_wtime();
#pragma omp parallel for private(i, j)
for(i=0;i<n;i++)
{
  for(j=0;j<n;j++) {
    vysledMatic[i][j]=mat[j][i];
  }
}

time_end = omp_get_wtime();
    printf(" %f sekund.\n", time_end-time_start);
free(mat);
free(vysledMatic);
return 0;
}
```

B Obsah CD

Složky:

- Ve složce C se nachází zdrojové soubory C projektu
- Ve složce OpenCL se nachází zdrojové soubory OpenCL projektů.
- Ve složce BP se potom nachází samotná práce v odt a pdf. Dále pak i přebal BP v pdf.