

**EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY**

Evidenčné číslo: 17300/B/2011/2470246218

GPU computing

Bakalárska práca

Bratislava 2011

Matej Hudák

**EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY**

GPU computing

Bakalárska práca

Študijný program: Hospodárska informatika

Študijný odbor: 9.2.10 Hospodárska informatika

Školiace pracovisko: Katedra aplikovanej informatiky

Školiteľ: Ing. Miroslav Kršjak, PhD.

Bratislava 2011

Matej Hudák

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Matej Hudák
Študijný program: Hospodárska informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.10 Hospodárska informatika
Typ záverečnej práce: Bakalárska záverečná práca
Jazyk záverečnej práce: slovenský

Názov: GPU computing
Anotácia: Práca sa zaoberá prehľadom a porovnaním technológií pre využitie GPU pri ekonomických výpočtoch

Vedúci: Ing. Miroslav Kršjak, PhD.
Katedra: KAI FHI - Katedra aplikovanej informatiky FHI
Vedúci katedry: doc. Ing. Gabriela Kristová, CSc.
Dátum zadania: 28.09.2010

Dátum schválenia: 19.10.2010
doc. Ing. Gabriela Kristová, CSc.
vedúci katedry

Čestné vyhlásenie

Čestne vyhlasujem, že záverečnú prácu som vypracoval(a) samostatne a že som uviedol všetku použitú literatúru.

Dátum:

.....
(podpis študenta)

Pod'akovanie

Chcel by som pod'akovať všetkým, ktorí mi akýmkoľvek spôsobom pomohli pri spracovaní tejto bakalárskej práce. Moje pod'akovanie patrí najmä vedúcemu práce, Ing. Miroslavovi Kršjakovi PhD., za vedenie a kolegom v práci, za poskytnutie kompatibilnej hardvérovej výbavy.

Abstrakt

HUDÁK, Matej: *GPU Computing*. – Ekonomická univerzita v Bratislave. Fakulta hospodárskej informatiky; Katedra aplikovanej informatiky – Vedúci záverečnej práce: Ing. Miroslav Kršjak, PhD. – Bratislava: FHI EU, 2011, 34

Cieľom záverečnej práce bolo priblížiť nový trend vo svete informačných technológií. Popísať jeho vývoj, problémy ktoré musel riešiť a súčasný stav.

Práca je rozdelená do 8 kapitol. Obsahuje 0 grafov, 0 tabuliek a 0 príloh. Prvá štyri kapitoly sa venujú vývoju GPU computing-u, čo inšpirovalo jeho vznik a ako sa vyvíjali hardvérové prostriedky v danej oblasti. V ďalšej časti sa charakterizuje softvérové a programovacie prostredie moderných GPU platforiem. Predstavujú sa kľúčové vlastnosti daných platforiem a zdrojové kódy.

Záverečná kapitola sa zaoberá ukážkami z reálneho použitia GPU aplikácií v IT svete.

Výsledkom riešenia danej problematiky je pomoc pri rozhodovaní sa, v akom prostredí by zaniatený programátor mohol začať vyvíjať svoje GPGPU aplikácie.

Kľúčové slová:

GPU computing, GPGPU, kernel, OpenCL, CUDA

Abstract

The aim of the thesis was to present a new trend in the world of information technology. Describe its evolution, problems which had to be solved and current status. The work is divided into 8 chapters. Contains 0 graphs, tables 0 and 0 attachments. The first four chapters are devoted to the development of GPU computing, in which inspired its creation and how the hardware resources developed in this area. The next section describes the software and programming environment of modern GPU platforms. Represent the key characteristics of the platforms and their source code. The final chapter deals with examples of real applications use the GPU in the IT world. The result of solving the issue is to help you decide in which environment an avid programmers would begin to develop their GPGPU applications.

Keywords:

GPU computing, GPGPU, kernel, OpenCL, CUDA

Obsah

Úvod.....	8
Vývoj GPGPU	9
Vývoj osobných počítačov – CPU (Central Processing Unit).....	9
Vývoj osobných počítačov – GPU (Graphic Processing Unit).....	10
GPU – Graphic processing unit	12
Architektúra grafických procesorov	12
Evolúcia GPU architektúry	13
Architektúra moderných grafických procesorov.	14
GPU Computing	16
Programovací model grafických procesorov	16
Všeobecné výpočty na grafický kartách	17
Programovanie grafických výpočtov na GPU.	17
Softvérové riešenia	18
NVIDIA CUDA.....	19
CUDA ako architektúra	19
CUDA C	20
Kernel funkcia.....	22
OpenCL.....	23
Modely paralelného programovania	24
Dátovo paralelný model.....	24
Úlohovo paralelný model.....	24
OpenCL – programovací jazyk.....	25
OpenCL verus CUDA	29
Súčasný GPGPU projekty	30
BOINC	30
SETI@home	31
Folding@home	32
Iné využitie GPGPU	33
Záver	35
Použité zdroje	36

Úvod

GPU computing predstavuje moderný trend v informatickom odvetví. Mnoho odborníkov alebo skúsených používateľov sa môže opýtať, čo stojí za tým ruchom okolo tejto novinky? Ved' paralelne spracovanie dát je známe už niekoľko rokov. Mnohé serverové stanice alebo vedecké super počítače využívajú množstvo procesorov ktoré umožňujú paralelné spracovanie dát . Na trhu je dostupných množstvo centrálnych procesorov rôznych architektúr, ktoré obsahujú niekoľko jadier a zvládajú aj paralelné výpočty na slušnej úrovni. Za zmienku určite stoja procesory architektúry POWER do spoločnosti IBM, hojne využívaných v Apple počítačových zostavách. Ďalej procesor architektúry CELL, od spoločností Toshiba a Sony, ktorý je celkom úspešne nasadený v hernej konzole Playstation 3. Výhodou grafických kariet je však ich vysoká kompatibilita a ľahká dostupnosť výpočtového potenciálu. Grafické karty sú len minimálne viazané na operačné systémy, čo jednoznačne zvyšuje ich dostupnosť. Vďaka tomu sa aj domáci počítač dokáže zmeniť na výkonné výpočtové stredisko. Ďalšou nespornou výhodou grafických kariet je, že tento hardvér je dostupnejší aj pre vývojárov a ich produkty môžu smerovať na rôzne segmenty trhu. Za zmienku určite stoja projekty distribuovaného spracovania dát, napríklad BOINC alebo SETI. Tie pomocou internetu, aplikačného rozhrania podporujúceho GPU computing a počítačových nadšencov po celom svete, dosiahli obrovský výpočtový výkon. A to až tak, že predbehli aj najlepšie super počítače na svete. Vďaka takýmto projektom možno už v blízkej dobe objavíme príčiny vzniku rakoviny (projekt Folding@home), alebo sa nám podarí kontaktovať mimozemské civilizácie.

V tejto práci sa chcem zamerať najmä na vývoj hardvéru, ktorý umožnil, že sa GPU computing dostal do dnešnej podoby. Ďalej sa sústredím na programovacie jazyky a štandardy ktoré sa snažia o čo najefektívnejšie využívanie výpočtového potenciálu grafických kariet. Porovnáam možnosti ktoré ponúkajú súčasné spoločnosti vyvíjajúce grafické procesory a aké sú ich výhody a nevýhody.

Vývoj GPGPU

Vývoj osobných počítačov – CPU (Central Processing Unit)

Za posledných 30 rokov bolo zvyšovanie pracovnej frekvencie jadra procesora jednou z najpoužívanejších metód na zvyšovanie výkonu osobných počítačov. Prvé počítače zo začiatku 80. rokov 20. storočia využívali procesory pracujúce na frekvencii 1 MHz (fyz. megahertz). Po tridsiatich rokoch sa frekvencie procesorov pohybujú v rozmedzí 1 GHz až 4 GHz, čo je viac ako 1000 násobok frekvencie procesorov prvých osobných počítačov. Aj keď zvýšenie frekvencie jadra procesora nepredstavuje jediný spôsob zvýšenia výkonu počítača, stále predstavovalo a predstavuje, spoľahlivý spôsob nárastu výkonu. Za posledných 10 rokov ale výroba silnejších, jednojadrových procesorov narazila na problémy. Neustále zvyšovanie pracovnej frekvencie jadra, vedie k veľkému odpadovému teplu z procesorov a k vysokej energetickej náročnosti. Taktiež kvôli zásadným obmedzeniam súčasných procesorových architektur, sa už pri výrobe integrovaných obvodov naráža aj na problémy s ďalším zmenšovaním tranzistorov.

Mimo sveta osobných počítačov, sa v odvetví super počítačov tiež používal rovnaký spôsob zvyšovanie výkonu. V porovnaní s nárastom výkonu samotných procesorov sa obrovský výkon super počítačov dosahoval hlavne zvyšovaním počtu samotných procesorov. Dnešné super počítače sa často skladajú z desiatok až stoviek procesorov pracujúcich v tandeme. Pri hľadaní ďalších možností zvýšenia výkonu sa vyskytla otázka: *Nebude lepšie miesto neustáleho zlepšovania výkonu jedného jadra procesora zapracovať viac jadier do jedného počítača?* V roku 2005 sa tak pod veľkým konkurenčným tlakom a len s malým množstvom iných alternatív začali na trhoch objavovať viacjadrové procesory. Na začiatku to boli dvojjadrové procesory, neskôr, keď sa táto cesta osvedčila ako správna, začali technologické giganty produkovať troj-, štvor-, šesť- až osem- jadrové procesory. *Niekedy označovaný ako multijadrová revolúcia, tento trend zaznamenal obrovský posun vo vývoji procesorov pre osobné počítače.* V dnešnej dobe je už problém zakúpiť počítač, ktorého procesor má len jedno jadro. Aj tie najlacnejšie procesory zvyčajne obsahujú aspoň dve jadrá. Popritom ešte poprední výrobcovia ohlásili plány na dvanásť až šestnásť jadrové procesory. Doba paralelných výpočtov začala.

Vývoj osobných počítačov – GPU (Graphic Processing Unit)

Masívnejšie rozšírenie počítačov medzi širokú verejnosť a do domácností začalo pred vývojárov klásť nové výzvy. Kým v odbornom prostredí nebola núdza o programátorov a správcov systémov, rovnaké podmienky neboli predstaviteľné v prostredí škôl netechnických smerov alebo domácnostiach. Užívatelia začali pre prácu s počítačom čoraz viac vyžadovať prívetivejšie užívateľské rozhranie (UI) ktoré ich malo odbremeniť od nutnosti dôkladnej znalosti kódu.

Keď sa v osemdesiatych rokoch dvadsiateho storočia začali objavovať prvé operačné systémy z grafickým užívateľským rozhraním, začali výrobcovia hardvéru ponúkať nový druh procesorov. Išlo o takzvané GPU (Graphic Processor Unit) – grafické procesory. Ich úlohou bolo odbremeniť centrálny procesor počítača od grafických výpočtov a zároveň zvýšiť ich rýchlosť. Dopyt na grafické procesory ale nebol len v sfére osobných počítačov ale aj pri profesionálnom využití. V tom období sa spoločnosť Silicon Graphics snažila o popularizáciu trojrozmernej grafiky na rôznych trhoch. Trojrozmerná vizualizácia si získavala uplatnenie v armáde, bezpečnostných úradoch, vedeckých a technických vizualizáciách a taktiež vo filmovom odvetví.

V roku 1992 spoločnosť Silicon Graphics otvorila programovacie prostredie k ich grafickému hardvéru publikovaním knižnice OpenGL. Spoločnosť sa snažila o presadenie OpenGL ako platformovo nezávislej metódy na programovanie trojrozmerných aplikácií.

Ďalším skokom vpred bolo vydanie grafickej karty S3 86C911 spoločnosťou Silicon Graphics. Išlo o prvú jedno čipovú grafickú kartu s podporou 2D akcelerácie. Grafická karta S3 86C911 značne inšpirovala konkurenciu a 2D akcelerácia sa začala objavovať v každej nasledujúcej vydannej grafickej karte. Ďalšou pákou na zlepšovanie architektúr grafických kariet bola stále prepracovanejšia podpora 2D akcelerácie na softvérovej úrovni. Spoločnosť Microsoft vydala knižnicu WinG pre Windows 3.x, neskôr nasledoval DirectDraw. Programovateľné aplikačné rozhranie DirectDraw spravuje hlavne 2D grafiku, prácu s oknami a má viaceré funkcie na nízkej úrovni (napr. priamy prístup do pamäte). DirectDraw sa neskôr stalo súčasťou programovateľného aplikačného rozhrania DirectX, dnes ale pod názvom Direct2D.

Spočiatku samostatnou kapitolou grafických kariet bola trojrozmerná akcelerácia, o tú sa najprv starali centrálné procesory počítačov. Veľkou inšpiráciou pre vývoj grafických kariet pre osobné počítače schopné 3D akcelerácie bol úspech herných konzol. Najmasovejšie

nasadenie 3D grafických akceleratorov sa objavilo v konzolách PlayStation a Nintendo 64. Prvé 3D grafické karty nezaznamenali veľký úspech, mnohé z nich sa stali doslova prepadákmi. Chýbajúca podpora pre 2D akceleráciu, kolízne stavy a problémy s komunikáciou s 2D grafickými čipmi neumožnili presadenia sa lacnejších 3D grafických kariet na trhu. Až čipom Vérité od spoločnosti Rendition Inc. sa podarilo spojiť 2D akceleráciu s 3D funkcionalitou do jedného čipu.

Na poli programovateľných aplikačných rozhraní sa začal rozbiehať konkurenčný boj medzi OpenGL a DirectX. DirectX, ako produkt spoločnosti Microsoft si na jednej strane získaval obľubu medzi vývojármi softvéru a hier pre operačný systém Windows, ale strácal kvôli viacerým obmedzeniam, striktným nárokom na hardvérové vybavenie grafických kariet a taktiež kvôli chýbajúcim funkciám. Napríklad OpenGL už od svojho počiatku umožňovala kontrolu tzv. T&L alebo TCL (z angl. Transform, clipping and lighting), teda transformácie, vykresľovania a nasvietenia. Tie boli do rozhrania DirectX implementované až v jeho súčasti Direct3D 7.

V auguste roku 1999 uviedla spoločnosť NVIDIA na trh grafickú kartu GeForce 256. O tomto momente sa dá hovoriť ako o prelomovom, nakoľko úspech tejto grafickej karty mal značne devastčné účinky na konkurenčné firmy. Karta svojou architektúrou zaznamenala viaceré prvenstvá. Bola prvou kartou s plnou podporou rozhrania Direct3D 7, prvou na trhu neprofesionálneho využitia ktorá mala plnú hardvérovú podporu T&L. Implementácia týchto funkcionalít slúžila ako základný kameň pre neskorší vývin pixelových a vektorových shaderov. V nasledujúcom období boli z hry vyradené viaceré spoločnosti a konkurenčný boj na trhu herných grafických kariet sa obmedzil na spoločnosti NVIDIA a ATI Technologies. Ostatné spoločnosti sa stiahli z tohto boja a zamerali sa buď na integrované grafické čipy alebo na profesionálne grafické karty.

Ďalšie roky sa niesli v konkurenčnom boji spoločnosti NVIDIA a ATI. NVIDIA uviedla prvú grafickú kartu s hardvérovou podporou rozhrania DirectX 8, no pri stále novších rozhraniach, menovite DirectX 9, DirectX 10 a DirectX 11 ich vždy predbehla spoločnosť ATI. Pozornosť odborníkov pritiahli niektoré z podmienok na architektúru grafickej karty s plnou hardvérovou podporou rozhrania DirectX 8. Konkrétne o plne programovateľné pixelové a vektorové shader-e. *Po prvýkrát tak vývojári získali určitú kontrolu na výpočtami rátanými na procesore grafickej karty.*

GPU – Graphic processing unit

Neustály dopyt spotrebiteľov po silnejších grafických kartách, 3D grafike vo vysokom rozlíšení a v reálnom čase spôsobili, že grafické procesory sa vyvinuli do úrovne vysoko paralelných, multivláknových, viac jadrových procesorov s obrovskou výpočtovou silou a veľkou priepustnosťou pamäťových radičov. Súčasné procesory grafických kariet porážajú s veľkým náskokom centrálné procesory osobných počítačov čo sa týka rýchlosti a objemu vykonaných inštrukcií za časovú jednotku. Ten síce už ponúkali aj staršie grafické karty, ale uzavretosť architektúry a slabšia programovateľnosť výpočtových jednotiek neumožňovala efektívne využívanie tohto potenciálu na iné, než grafické výpočty. Súčasný trend vývoja grafického hardvéru sa však zameria na otvorenie architektúr a sprístupnenie výpočtových kapacít pre programátorov. Za posledné roky, konkrétnejšie od vydania grafickej karty Nvidia GeForce 256 v roku 1999, sa grafické procesory vyvinuli z funkčne pevne špecializovaného procesora na plne rozvinutý, programovateľný, paralelný procesor. Samozrejme, zo zachovaním všetkých pôvodných funkcií a účelu použitia. Do popredia sa ale stále viac dávajú programovateľné aspekty grafických procesorov.

Architektúra grafických procesorov

Grafická pipeline

Vstupnými dátami pre grafické karty sú zoznamy číselných údajov, ktoré predstavujú súradnice objektov v 3D priestore. Najčastejšie sa jedná o súradnice trojuholníkov. Tieto číselné údaje sú spracované v postupnosti viacerých krokov, mapované na obrazovku, kde sa z nich vytvorí hotový obrazec. Pre lepšiu predstavu a pochopenie uvediem v nasledujúcej časti niektoré zo základných operácií na grafických procesoroch.

Vertexové operácie – vstupné súradnice pre grafické karty sa odvodzujú z vrcholov, vertexov, jednotlivých trojuholníkov. Každý vertex sa transformuje do trojrozmerného priestoru na obrazovke a podlieha tieňovaniu. Tieňovanie každého vertexu závisí od zdrojov osvetlenia pre momentálne renderovanú scénu. Práve tento dopyt po čo najvernejšej grafike s realistickým osvetlením sa môže označovať za hlavnú príčinu toho, že grafické procesory sa vyvinuli na masívne paralelné výpočtové jednotky. Je úplne bežné, že takéto grafické scény sa skladajú z desiatok až stoviek tisícov vertexov v jednej snímke. A keďže každý vertex sa

počíta nezávisle a v reálnom čase, je paralelný procesor nutnosťou.

Skladanie vrcholov – skladanie vertexov do trojuholníkov je dnes základnou, hardvérovo urýchlňovanou funkciou grafických procesorov.

Rastrovanie - predstavuje proces, pri ktorom sa rozhoduje o tom, ktorý pixel výstupnej obrazovky je pokrytý daným polygónom. Každý takýto polygón generuje premennú nazývanú fragment, na každej pixel pozícii ktorú prekrýva. Nakoľko sa však na pozícii jedného pixela môžu prekrývať viaceré polygóny, farba jednotlivého pixela tak môže byť vypočítaná z viacerých fragmentov.

Fragmentové operácie – každý fragment podlieha tieňovaniu do finálnej farby na základe údajov o farbách v každom vertexe. Taktiež sa ešte používajú doplňujúce informácie z pamäte, ako napríklad textúra pre daný grafický objekt. Tak ako pri výpočtoch vertexov polygónov, aj táto fáza je počítaná paralelne. Zároveň predstavuje výpočtovo najnáročnejšiu fázu grafického spracovania.

Kompozícia – fragmenty sú nakoniec skladané do finálneho obrazu, s jednou farbou na jeden pixel. Netreba pripomínať, že pri dynamických scénach, čo sú dnes skoro všetky, je tieto operácie vykonávať pri každej zmene scény. Pri súčasných hrách, ktoré využívajú rôzne grafické vylepšenia, je nutné pre zachovanie plynulého pohybu, vyrábať v jednej sekunde 30 až viac snímok. To pri rozlíšení 1920 na 1080 bodov a 30 snímkach za sekundu predstavuje 62 208 000 pixelových operácií. Samozrejme, všetky paralelne. Tieto operácie dostupné na vrcholoch a fragmentoch boli konfigurovateľné, ale nie programovateľné. To znamená, že programátor môže kontrolovať len pozíciu farieb a svetelných zdrojov, ale nemôže ovládať model nasvietenia, ktorý je potrebný pri interakcii objektov.

Evolúcia GPU architektúry

Uzavretá architektúra spracovania dát v procesoroch grafických kariet neumožňovala, aby sa dali efektívne programovať komplikované shader a svetelné operácie, ktoré sú potrebné pre komplexné grafické efekty. Prelomovým okamihom bolo práve otvorenie architektúry a umožnenie plnej kontroly operácií na vertexoch a fragmentoch. Od otvorenia architektúr sa programy spravujúce výpočty fragmentov a vertexov značne vyvinuli a ponúkajú funkcionality na vyššej úrovni. Patrí tu napr. vyšší limit na veľkosť vstupných údajov, vyššie limity na výpočtové zdroje a čoraz širšia inštrukčná sada spolu s flexibilnou kontrolou riadenia operácií. Kým spočiatku boli tieto inštrukčné sady oddelené pre vertexové

a fragmentové operácie, v súčasnosti grafické karty podporujú unifikovaný Shader Model 4.0. Ten nahradil doteraz používané Vertex a Fragment shadre. Shader Model 4.0 si ale kladie určité podmienky na hardvér, aby mohol byť využívaný. Patria tu:

- hardvér musí podporovať shader programy sa aspoň 65k statických inštrukcií a neobmedzený počet dynamických inštrukcií.
- inštrukčná sada musí zvládať podporu 32 bitových celých čísel a 32 bitových čísel s pohyblivou desatinou čiarkou.
- hardvér musí umožňovať ľubovoľné prístupy na čítanie, priame alebo nepriame, z globálnej pamäte.
- dynamická kontrola toku dát musí byť podporovaná v podobe slučiek aj vetiev.

Dá sa vlastne povedať, že kým grafické karty v minulosti umožňovali len fixné riešenie zreteľovaných úloh, s malou mierou programovateľných súčastí, dnešné grafické procesory predstavujú vysoko programovateľné procesory. Samozrejme, stále s využitím jednotiek s fixnou funkcionalitou.

Architektúra moderných grafických procesorov.

Ako už bolo spomínané, procesory grafických kariet sú primárne určené na spracovanie iných požiadaviek, ako sú kladené na centrálné procesory. Od grafických kariet sa vyžaduje vysoká miera paralelnosti výpočtov, s dôrazom na vysokú dátovú priepustnosť. Vďaka týmto požiadavkám, grafické procesory napredujú v inom smere ako centrálné procesory. Tým najpodstatnejším rozdielom je práve paralelné spracovanie dát. Súčasné grafické procesory podporujú viacero modelov paralelného spracovania dát. Medzi najvýraznejšie z nich patrí dátovo-paralelný programovací model a úlohovo-paralelný programovací model. Ďalšou vlastnosťou grafických procesorov je, že úlohy určené na paralelné spracovanie delí v priestore a nie v čase. Centrálny procesor si vezme určitý článok zreteľovaných inštrukcií, vykoná na nich požadované operácie a ak dôjde k jej úspešnému ukončeniu, procesor môže pokračovať v riešení ďalšieho článku. Centrálny procesor teda delí inštrukcie v čase, s využitím všetkých dostupných prostriedkov na výpočet jednej časti inštrukcií v jednom okamihu. Grafický procesor však uplatňuje iný prístup. Dostupné výpočtové prostriedky si delí podľa jednotlivých článkov zreteľovaných inštrukcií. Tak teda časť procesora pracujúca na jednom článku, po jeho úspešnom ukončení, vracia svoj výstup ďalšej časti procesora. Inštrukcie sú teda delené v priestore. Inštrukčná sada, ktorú využívali ešte fixné

procesory grafických kariet, dosahovala veľmi vysokú účinnosť. Grafický procesor totiž dokázal na každom článku inštrukcií vykázat dátovo-paralelný model spracovania dát na jednej úrovni. Teda spracovať viaceré elementy v rovnakom čase. Zároveň však uplatňoval úlohovo-paralelný model, nakoľko na rozličných elementoch sa v rovnakom čase, vykonávali rozdielne inštrukcie. Ďalší nárast výkonu dosahovali grafické karty tým, že na každú špecifickú úlohu vedeli využiť špecifickú časť hardvéru.

GPU pipeline sa tak rozčlenila do viacerých úrovní, pričom každá mohla byť akcelerovala špeciálnou časťou hardvéru. V centrálnom procesore, môže požadovaná operácia, medzi vstupom a výstupom z neho, prejsť cez dvadsať cyklov. No v grafickom procesore môže prejsť aj cez tisíce cyklov od začiatku po koniec. Výsledkom toho je síce väčšie oneskorenie výpočtu, no vysoká úroveň paralelnosti medzi jednotlivými úrovňami zabezpečuje vysoký objem spracovaných dát.

Najväčšou nevýhodou úlohovo-paralelného modelu grafických procesorov je riadenie vyťaženia. Výkon GPU pipeline, ako každej inej, je závislý od najpomalšej úrovne. Ak vertex program vykazuje väčšiu komplexnosť ako fragment program, výsledný výstup závisí od výkonu vertex programu. Na začiatku sa programovateľné úrovne vertex a fragment shaderov líšili, takže jednotlivé úrovne boli oddelené. Postupne sa však viedli viaceré kroky k ich zjednoteniu. Spolu s vydaním platformy DirectX 10 sa už objavuje unifikovaná shader architektúra, ktorá spája viaceré programovateľné jednotky do jedného celku. Tento unifikovaný shader podporuje aj dátovo-paralelný aj úlohovo-paralelný model. Zjednotením shaderov sa dosiahlo, že programátori sa môžu priamo zamerať na programovanie pre jednu jednotku. Nemusia sa už zaoberať delením kódu medzi viacero hardvérových jednotiek.

GPU Computing

Programovací model grafických procesorov

Programovateľné jednotky grafických procesorov využívajú SPMD programovací model. SPMD (z angl. SINGLE-PROGRAM MULTIPLE-DATA, jeden program, zložené dáta.). Na zvýšenie výkonu, grafický procesor spracúva viacero elementov, vrcholy alebo fragmenty, paralelne jedným programom. Elementy sú od seba nezávislé a v základnom modeli nemôžu medzi sebou komunikovať. Každý GPU program musí byť štruktúrovaný týmto spôsobom:

- veľa elementov, každý spracovaný paralelne, jedným programom.
- elementy môžu pracovať s 32 bitovým celým číslom, alebo s desatinným číslom s pohyblivou čiarkou, nad kompletnou, všeobecnou inštrukčnou sadou.
- elementy dokážu čítať zo zdieľanej pamäte (operácia „gather“). Pri najnovších grafických kartách už zvládajú aj zápis do ľubovoľných oblastí v zdieľanej, globálnej pamäti (operácia „scatter“).

Tento programovací model je vhodný pre lineárne programy, kde veľa elementov môže byť spracovaných v krokoch, s nemenným kódom. Takto napísaný kód predstavuje model SIMD (z angl. SINGLE INSTRUCTION, MULTIPLE DATA, jednotné inštrukcie, zložené dáta.). Nakoľko sa však shader programy dostali na vyššiu úroveň, začali programátori preferovať iné možnosti. Dovolili rozličným elementom používať rozdielne spôsoby spracovania v jednom programe. To viedlo k všeobecnejšiemu SPMD modelu. Jednou z výhod GPU je schopnosť rozdeľovať výpočtové zdroje a rozdielny spôsob kontroly. Ak sa totiž umožní, aby každý element podliehal rozdielnemu spôsobu spracovania, vyžaduje si to značné nároky na kontrolu hardvéru. Moderné grafické procesory však podporujú ľubovoľný prístup ku kontrole priebehu výpočtu, výmenou za zákaz nesúvislého vetvenia výpočtov. Elementy sa tak zoskupujú do blokov a bloky sú spracovávané paralelne. Ak dôjde k vetveniu v rámci bloku, hardvér si vynúti vypočítanie obidvoch vetiev. A to nie len pre element bloku, ktorý to spôsobil, ale pre všetky elementy bloku. Veľkosť bloku sa označuje ako „branch granularity“. V súčasnosti je stanovená na 16 elementov. Pri písaní GPU programov sú tieto bloky povolené, ale ich využívanie nie je neobmedzené.

Všeobecné výpočty na grafický kartách

Presunutie všeobecných výpočtov na grafický procesor využíva hardvér rovnakým spôsobom, ako štandardná grafická aplikácia. Predsa sú však medzi týmito procesmi isté rozdiely. Skutočné operácie sú síce rovnaké, problém je však v tom, že terminológia využívaná pri všeobecných výpočtoch sa líši od tej, ktorá sa využíva pri grafických výpočtoch

Programovanie grafických výpočtov na GPU.

Programátor špecifikuje geometrický obrazec, ktorý pokrýva istú časť obrazovky. Rasterizér generuje fragment na každej pozícii pixla pokrytého geometrickým obrazcom.

- Každý fragment podlieha shadingu vo fragment shaderi.
- Fragment shader vypočíta farebnú hodnotu pre každý fragment. Tá závisí od viacerých premenných, či už nasvetlenia alebo textúry z pamäte.
- Výsledný obraz môže byť ďalej používaný ako textúra pri ďalších operáciách s obrazcom.

Programovanie všeobecných výpočtov na GPU

Jeden z najťažších problémov programovania GP GPU aplikácií je predurčenie grafických procesorov na grafické výpočty. Čiže aj keď všeobecné výpočtové úlohy nemali nič spoločné s grafikou, museli sa programovať cez aplikačné rozhrania pre grafické karty. K tomu ešte museli byť aj úlohy štruktúrované vo forme grafickej pipeline. To bolo zapríčinené tým, že určité operácie sú dostupné len v určitom poradí. Nedá sa k nim pristupovať priamo, čo by každý programátor preferoval. Toto obmedzenia už ale výrobcovia grafických procesorov odstránili a dnes sú GPU aplikácie organizované v nasledujúcej štruktúre.

- Programátor priamo definuje ktoré programové vlákna majú byť vypočítané na grafickej karte.
- Hodnota každého vlákna je potom vyrátaná všeobecným programom využívajúcim SPMD model.
- Hodnota pre každé vlákno je vypočítaná ako kombinácia viacerých matematických operácií. Taktiež sú zapojené do výpočtu „gather“ a „scatter“ operácie na globálnej

pamäti. Na rozdiel od predchádzajúcej metódy, rovnaký buffer môže byť použitý ako na zápis tak aj na čítanie. Umožňuje to využívanie flexibilnejších algoritmov ktoré šetria pamäť.

- Výsledný buffer na globálnej pamäti môže byť používaný ako vstup na ďalšie výpočty.

Tento programovací model je veľmi výkonný z viacerých dôvodov. Umožňuje priamo v programe špecifikovať dátovo-paralelný model a hardvér ho potom dokáže náležite použiť. Ďalej kladie vysoký dôraz na vyváženosť medzi všeobecnosťou a obmedzeniami, za účel dosiahnutia výkonu. Obmedzenia také, ako boli spomínané pri SPMD modely. Teda obmedzenie vetvenia, komunikácie medzi elementmi a podobne. Nakoniec je to priamy prístup k programovateľným jednotkám grafického procesora. Eliminuje sa tak nutnosť maskovania všeobecných výpočtov za grafické. Výsledkom to sú programy, ktorých zdrojové kódy sú viac podobné klasickým programom. S tým súvisí jednoduchšie spracovanie kódu a debugovanie. Spolu s čoraz sofistikovanejšími nástrojmi a kompilátormi, je čoraz jednoduchšie plne využívať potenciál grafických procesorov na všeobecné výpočty.

Softvérové riešenia

V minulosti sa väčšina GPGPU programovania riešila priamo cez aplikačné rozhranie grafických kariet. Aj keď sa mnohým vývojárom tak podarilo úspešne naprogramovať fungujúce aplikácie, ich programová logika nebola veľmi ideálna. Vývojári používali fixne určené súčasti grafických kariet, na GPGPU operácie. Problémom v tomto prípade bolo, že akýkoľvek všeobecný výpočet sa musel maskovať za grafický. V istom ohľade sa táto situácia zlepšila po uvedení plne programovateľných fragment procesorov. Tie už ponúkali iste pseudo jazyky, stále však boli veľkou príťažbou pre programátorov. Postupne sa však začali objavovať programovacie jazyky vyššej úrovne určené na programovanie shaderov. Microsoft predstavil spolu s rozhraním DirectX 9 jazyk HLSL („high-level shading language“), využívajúci rozhranie jazyka C. V podobnom smere sa vybrala NVIDIA s ich jazykom Cg. Ten ponúka podobnú funkcionality ako HLSL jazyk, je však lepšie orientovaný na hardvér. Štandard OpenGL využíva ne tieto účely jazyk GLSL. Stále sú to však len jazyk primárne určené na programovanie grafických výpočtov. Teda všetky výpočty musia byť vyjadrené vo forme grafických termínov, textúry, fragmenty a podobne. Hoci umožnili všeobecnejšie

programovanie grafických kariet, stále neboli veľmi prístupné bežným programátorom. Projektov na odstránenie týchto prekážok bolo viacero, no v dnešnej dobe sa presadili najmä projekty CUDA od spoločnosti NVIDIA a projekt Fire Stream od spoločnosti ATI. Treba podotknúť, že tieto projekty boli spustené technologickými lídrami v oblasti vývoja grafických procesorov. Ich veľkou výhodou je, že kým ostatné projekty a platformy stavali už len na hotovom hardvéri, tieto spoločnosti si architektúru grafických čipov mohli upraviť podľa vlastných potrieb.

NVIDIA CUDA

V novembri roku 2006 sa na trh dostala grafická karta GeForce 8800 GTX od spoločnosti NVIDIA, ktorá bola ako prvá navrhnutá s CUDA architektúrou. Tá bola zároveň aj prvá grafická karta od NVIDIA, ktorá mala hardvérovú podporu pre nový štandard DirectX 10. Pár mesiacov po vydaní grafickej karty GeForce 8800 GTX bol predstavený kompilátor pre jazyk CUDA C. CUDA C predstavuje štandard jazyka C rozšírený o kľúčové slová, ktoré umožnili naplno využiť grafické karty na rávanie všeobecných výpočtov.

CUDA ako architektúra

CUDA kompatibilné grafické karty už obsahujú *unified shader pipeline*, ktorá umožňuje, aby každá aritmeticko-logická jednotka (ALU) na čipe bola spravovaná priamo kódom CUDA kompatibilného programu. Takto napísaný program potom bez nutnosti kamufláže dokáže odbremeniť od výpočtov centrálny procesor a presunúť výpočty na ALU jednotky grafickej karty. Tieto aritmeticko-logické jednotky boli navrhnuté aby spĺňali IEEE (skratka od **Institute of Electrical and Electronics Engineers**) štandard pre výpočty s desatinou čiarkou s jednoduchou presnosťou.

Na rozdiel od predchádzajúcich architektúr grafických kariet, architektúra CUDA bol navrhnutá na využívanie inštrukčnej sady určenej na všeobecné výpočty. Takáto zmena architektúry odstránila nutnosť maskovania výpočtov a zároveň aj zlepšila možnosti kontroly výsledkov a ich ďalšieho spracovania. Ďalšia zmena umožnila riadiacim jednotkám na grafickej karte ľubovoľný prístup k zápisu a čítaniu na pamäti a taktiež prístup do softvérovo riadenej cache pamäte. Tieto zmeny v architektúre grafického procesora umožnili aby CUDA

grafické karty dosahovali obrovský výkon či už v grafických výpočtoch alebo vo všeobecných výpočtoch.

CUDA C

Jazyk CUDA C bol vyvinutý za účelom plného využitia CUDA architektúry. Bol prvým programovacím jazykom vyvinutým GPU spoločnosťou. Kompilátor pre jazyk bol spolu s jazykom vydaný v marci roku 2007. Spolu s vydaním V tom čase už na spotrebiteľskom trhu boli bežne dostupné CUDA kompatibilné grafické karty.

Základné prvky CUDA kódu popíšem na malom príklade. Jedná sa o sčítanie hodnôt dvoch celých čísel. Výpočet je ale ponechaný na grafickú kartu.

```
#include "../common/book.h"

__global__ void add( int a, int b, int *c ) {
    *c = a + b;
}

int main( void ) {
    int c;
    int *dev_c;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, sizeof(int) ) );

    add<<<1,1>>>>( 2, 7, dev_c );

    HANDLE_ERROR( cudaMemcpy( &c, dev_c, sizeof(int),
                               cudaMemcpyDeviceToHost ) );
    printf( "2 + 7 = %d\n", c );
    HANDLE_ERROR( cudaFree( dev_c ) );

    return 0;
}
```

Ako prvé si môžeme všimnúť hlavičkový súbor **book.h**. Je súčasťou CUDA SDK a umožňuje volanie **kernel** funkcií.

Ďalej vidíme definovanú **kernel** funkciu. Odlíšenie **kernel** funkcie od klasických C funkcií sa ošetruje využitím kľúčového slova **__global__**. Kompilátoru sa tak dáva na vedomie, že daná funkcia sa má vykonať na dostupnom CUDA zariadení. **Kernel** funkcia má rovnaké vlastnosti ako klasické funkcie. Dokáže teda prijímať argumenty a vracať hodnotu. V našom prípade sú to dve celo číselné premenné a jeden smerník na celé číslo.

Funkcia **main** je C funkciou, nemá veľký zmysel ju bližšie popisovať. Zaujímavejšia je pre nás funkcia **HANDLE_ERROR**. Slúži na ošetrenie kolíznych stavov, ktoré môžu nastať pri operáciách s pamäťou grafickej karty. V našom prípade je to funkcia **cudaMalloc**.

cudaMalloc slúži na alokovanie pamäte na CUDA zariadení, teda kompatibilnej grafickej karte. V našom prípade sa na grafickej karte alokuje dostatočné miesto pre smerníkovú premennú **dev_c**. Ak alokácia zbehne bez problémov, program pokračuje vo vykonávaní kódu. V prípade chyby, je vykonávanie zastavené a zobrazená príslušná chyba.

Volanie **kernel** funkcií sa od klasických funkcií líši. Dôležitou súčasťou volania **kernel** funkcie sú ostré zátvorky <<<...>>> . Číselné hodnoty vo vnútri zátvoriek udávajú informácie o paralelnom spustení kódu. Volanie **kernel** funkcie s hodnotami <<< N, 1 >>> spôsobí, že funkcia je spúšťaná v N blokoch, teda kópiách toho istého kódu. Informáciu o číselnom označení spusteného bloku dostaneme týmto spôsobom: **int tid = blockIdx.x;** .

Ak však **kernel** zavoláme týmto spôsobom: <<< 1, N >>>, dochádza k spusteniu len jedného bloku ale v N vláknach. Tu sa číselné označenie vlákna získa pomocou **threadIdx.x**.

Obe tieto hodnoty sú užitočné, ak sa na grafickej karte pracuje napríklad s vektormi. Inkrementácia jedného prvku sa nedeje postupne, číslo za číslom, ale paralelne vo viacerých vláknach alebo blokoch. Poradie prvku sa určí číslom aktuálneho bloku, prípadne vlákna, a tak sa docieli nezávislosť jednotlivých výpočtov.

Ak **kernel** funkcia ukončí svoju činnosť bez problém. Smerníková premenná by mala obsahovať hodnotu vypočítaného čísla. Nakoľko však k výpočtu došlo na grafickej karte a **kernel** bol volaný z host funkcie, je nutné prekopírovať hodnotu do operačnej pamäte. Využíva sa tu opäť ošetrenie kolízneho stavu funkciou **HANDLE_ERROR**. Samotná funkcia na kopírovanie už v tomto prípade je označená ako **cudaMemcpy**. To, že sa jedná o kopírovanie z device pamäte na host pamäť ošetrujeme pridaním argumentu **cudaMemcpyDeviceToHost**.

Ďalej už len dochádza k vytlačeniu získanej hodnoty. Treba však podotknúť, že nakoľko jazyk C a ani CUDA C, neobsahujú **garbage collector**, je nutné uvoľniť alokovanú pamäť na zariadení. To sa ošetruje funkciou **cudaFree**.

Kernel funkcia

Na záver už len malý dodatok ohľadom **kernel** funkcií. Ako som už spomínal, kompilátoru sa kernel funkcia predstaví kľúčovým slovom **__global__**. Nie je to však jediný spôsob ako identifikovať **kernel** funkciu. Poznáme ešte ďalšie dva.

__global__ - môže byť volaná z host zariadenia, vykoná sa na device zariadení.

__device__ - môže byť volaná len z device zariadenie. Vykoná sa len na device zariadení.

__host__ - volaná z host zariadenia a vykonaná na host zariadení.

Hlavným rozdielom oproti jazyku C je, že CUDA C umožňuje jednoduché oddelenie kódu určeného na spracovanie centrálnym procesorom, od toho, ktorým sa má zaoberať grafický procesor. Spolu s pridaním viacerých funkcií, ktoré sú príznačné skôr pre jazyky vyššej úrovne, sa CUDA C stáva príťažlivým prostriedkom na tvorbu GP GPU programov.

No treba však podotknúť, že CUDA kód je určený primárne pre CUDA zariadenia. Stráca sa tak prenosnosť tohto kódu a zaťažuje sa podmienkou vlastníctva určitého druhu hardvéru. Tu nastupuje platforma OpenCL, ktorej hlavnou snahou je odstrániť závislosť na hardvéri.

OpenCL

OpenCL je aplikačné programové rozhranie pre písanie paralelného kódu. Je multiplatformové a založené na jazyku C. Umožňuje tak vývoj prenosných aplikácií, ktoré sú schopné využívať paralelné spracovanie na rôznom hardvéri. Vývoj OpenCL bol motivovaný potrebou štandardizovanej platformy na vývoj paralelných aplikácií. Sústredil sa najmä na odstránenie obmedzení, ktoré sprádzali používanie grafických rozhraní na tvorbu GPGPU programov. Modely paralelného programovania pre procesory sú zvyčajne založené na štandardoch ako OpenMP. Nezahŕňajú však možnosti využívania odlišných typov pamätí alebo podporu SIMD modelov. Heterogénne modely paralelného programovania, ako uviedla napríklad CUDA, dovoľujú využívať SIMD modely aj kompletnú správu pamätí, sú však viazané na platformu alebo na konkrétny typ hardvéru. Toto obmedzenie neumožňuje softvérovým vývojárom produkovať multiplatformové programy.

OpenCL pôvodne vyvíjal Apple Inc., k jeho zrodu prispeli aj spoločnosti AMD, IBM, Intel a NVIDIA. Technické špecifikácie štandardu OpenCL verzie 1.0 boli dokončené v novembri 2008. Po kontrole členmi Khronos Group boli technické špecifikácie schválené a publikované 8. decembra 2008. OpenCL využíva podobné princípy ako CUDA, odstraňuje však závislosť na konkrétnej architektúre hardvéru. Využíva totiž komplexnejšiu platformu a lepšiu správu zariadení, čo sa odráža v lepšej podpore prenosnosti medzi platformami a hardvérom. Implementácie štandardu OpenCL sú už prítomné na grafických kartách od AMD a NVIDIA, taktiež aj na centrálnych procesoroch architektúry x86. Dá sa očakávať, že tento štandard sa rozšíri aj na iné zariadenia napríklad DSP (Digital Signal Processor) alebo FPGA procesory (Field-programmable gate arrays). Vysoká prenosnosť však so sebou prináša aj isté obmedzenia. OpenCL programy sa musia lepšie pripraviť na rozmanitosť hardvéru. To potom vedie k väčšej komplexnosti kódu a spolu s tým aj zložitosti. Taktiež, keďže OpenCL ponúka aj veľké množstvo voliteľných funkcionalít, musí ošetriť to, že daný typ hardvéru nebude mať ich podporu. Je teda možné, že výkon prenosného OpenCL programu sa bude značne líšiť na rôznych zariadeniach. Z toho vyplýva, že na dosiahnutie rovnakého výkonu, bude prenosná aplikácia využívať sofistikované testy, na odhalenie dostupných funkcionalít. Tak si môže vybrať z viacerých možných kódov na spustenie, aby sa dosiahol čo najlepší výkon na momentálne využívanom zariadení.

Modely paralelného programovania

Paralelnosť výpočtov v počítačovom svete znamená, že viacero výpočtov môže byť vykonávaných v jednom momente. Či už ide o možnosť spustenia dvoch nezávislých procesov na centrálnom procesore, alebo o obrovské siete počítačov zapojených do tzv. „cloud computing“ projektov. Stále sa však vychádza z jedného základu a naráža sa na podobné problémy a prekážky.

OpenCL aj CUDA využívajú rovnaké modely paralelného programovania. Jedná sa o dátovo paralelný model a úlohovo paralelný model.

Dátovo paralelný model

Dátovo paralelný programovací model definuje výpočty ako sekvencie inštrukcií, ktoré sa vykonávajú na viacerých elementoch dát v pamäti. Tieto elementy sa nachádzajú v indexovanom priestore, ktorý určuje, ako sa budú dane elementy mapovať na „work-item“. Tento „work-item“ (z angl. pracovný nástroj) je ekvivalentom vlákna ktoré sa využíva v jazyku CUDA. OpenCL programovací model nevyžaduje, aby každý element pamäte, bol priradený práve jednému „work-item“, ktorý bude podliehať paralelnému spracovaniu dát.

OpenCL dátovo paralelný model je hierarchický. Jeho usporiadanie môže byť definované dvoma spôsobmi:

- Explicitne – vývojár definuje maximálny počet „work-items“, ktoré sa budú spracovávať paralelne. Taktiež definuje ich rozdelenie do špecifických pracovných skupín.
- Implicitne – líši sa tým, že rozdelenie do pracovných skupín je spravované priamo OpenCL. Vývojár len definuje maximálny počet „work-items“.

Úlohovo paralelný model

V tomto modeli, sú *kernel* inštancie vykonávané nezávisle od indexovaného priestoru. Paralelizmu sa prejavuje na vektorových premenných implementovaných OpenCL zariadením (device), zorad'ovaní paralelných úloh a na zorad'ovaní kernelov, prislúchajúcich prostrediu OpenCL.

OpenCL – programovací jazyk

Programovací jazyk OpenCL je postavený na jazyku C99. Podobne ako jazyk CUDA, zavádza niekoľko nových funkcionalít, nové kľúčové slová, ale tiež isté obmedzenia. OpenCL nemá podporu pre rekurziu, polia s variabilnou dĺžkou, smerníky na funkcie a bitové polia. Podporované nie sú ani hlavičkové súbory jazyka C99. Podporované sú OpenCL a C hlavičkové súbory. Na druhej strane je však jazyk rozšírený o funkcionality, ktoré uľahčujú písanie paralelných programov, prácu s vektorovými premennými. OpenCL taktiež umožňuje lepšiu synchronizáciu medzi paralelnými procesmi a poskytuje funkcie na prácu so skupinami alebo tzv. „work-items“.

Na nasledujúcom príklade ukážem volanie kernel funkcie v jazyku OpenCL. To je možné spraviť dvoma spôsobmi. Kernel si môžeme definovať ako klasickú funkciu spôsobom bežným pre programovacie jazyky. Tento kernel má za úlohu vypočítanie druhej mocniny čísel z poľa **input**.

```
__kernel void square( __global float* input, __global float* output,  const
unsigned int count)
{
    int i = get_global_id(0);
    if(i < count)
        output[i] = input[i] * input[i];
}
```

Treba si hneď všimnúť základný rozdiel medzi OpenCL a jazykom CUDA. Kým v CUDA sa na označenie kernel funkcie využívalo kľúčové slovo **__global__**, v OpenCL sa kernel funkcie označuje pridaním kľúčového slova **__kernel** pre dátový typ funkcie. Aj tu sa síce objavuje kľúčové slovo **__global**, tu má však odlišné uplatnenie. V prípade OpenCL ide o označenie pamäťového priestoru, v ktorom sú premenné alokované. OpenCL totiž tak ako CUDA ponúka možnosť využívať rôzne typy pamäte.

Na tejto kernel funkcii si môžeme všimnúť, že výsledok sa nevracia priamo cez príkaz return, ale pomocou smerníkovej premennej **output**. Opäť ako pri jazyku CUDA, tento spôsob uľahčuje ošetrenie kolíznych stavov, ktoré môžu nastať pri kopírovaní údajov z pamäte zariadenia do operačnej pamäte počítača.

Funkcia **get_global_id()**; v tomto prípade slúži na získanie globálneho indexu „work-item“ s ktorým sa momentálne pracuje.

Ďalší spôsob napísanie kernel funkcie spočíva v tom, že funkcia je napísaná ako textový reťazec.

```
const char *KernelSource = "\n" \
__kernel void square(                                     \n" \
    __global float* input,                               \n" \
    __global float* output,                             \n" \
    const unsigned int count)                             \n" \
{                                                         \n" \
    int i = get_global_id(0);                             \n" \
    if(i < count)                                         \n" \
        output[i] = input[i] * input[i];               \n" \
}                                                         \n" \
"\n";
```

Pri pohľade na tento kód sa môže vyskytnúť viacero otázok. Keď je funkcia napísaná ako textový reťazec, ako potom dochádza k jej volaniu? Volanie kernel funkcií sa nedeje priamo, ako napríklad v jazyku CUDA, ale pomocou ďalších funkcií.

Najprv je potrebné si vytvoriť kontext, v ktorom sa bude pracovať.

```
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
```

Premenná **device_id** je premennou jazyk OpenCL. Získava sa pomocou funkcie **clGetDeviceIDs()**. Táto funkcie preberá viacero argumentov, tie záležia od toho, na akom zariadení sa snažíme o vykonanie kódu. Po vytvorení kontextu sa volá ďalšia funkcia, ktorá vytvorí príkazy pre práve vytvorený kontext.

```
commands = clCreateCommandQueue(context, device_id, 0, &err);
```

Z tohto príkazu sa dá dedukovať, že úspešnosť vytvorenia jednotlivých OpenCL komponentov je nadväzná. Totiž ak by sa nám kontext nepodarilo vytvoriť a pokúsili by sme sa o volanie funkcie na vytvorenie príkazov, došlo by k pádu aplikácie. Preto je nutné, po každom takomto volaní OpenCL funkcií, overiť úspešnosť vytvorenia OpenCL premenných pomocou podmienky. Ak bude vyhodnotená ako nepravdivá, dôjde k ukončeniu programu.

Po vytvorení kontextu a príkazov môžeme zavolať funkciu, ktorá vytvorí OpenCL program.

```
program = clCreateProgramWithSource(context, 1, (const char **) &
KernelSource, NULL, &err);
```

Funkcie preberá ako argumenty vytvorený kontext a smerník na textový reťazec, ktorý obsahuje kód kernel funkcie. Ostatné argument slúžia na ošetrovanie kolíznych stavov. Taktiež, aj po tejto funkcii je nutné, aby sme overil je úspešné volanie. V prípade, že funkcia bol

napísaná klasickým spôsobom, teda nie ako reťazec, preberá sa ako argument referencia na danú funkciu. V tomto prípade by sa kód **&KernelSource** nahradil **& square**.

Ak bol program vytvorený, treba ho následne zložiť.

```
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```

Návratová funkcia hodnoty sa priradzuje do premennej **err**, ktorá slúži na overenie úspešného zloženia programu. Ak je program zložený, môžeme z neho vytvoriť kernel objekt.

```
kernel = clCreateKernel(program, "square", &err);
```

Po vytvorení kernel objektu, alokujeme pamäť na OpenCL zariadení. V tomto prípade sa bude jednať o pamäť premenných na vstupe aj na výstupe.

```
input = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * count,
NULL, NULL);
output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * count,
NULL, NULL);
```

Nasleduje zapísanie doteraz vytvorených objektov a premenných do pamäte OpenCL zariadenia.

```
err = clEnqueueWriteBuffer(commands, input, CL_TRUE, 0, sizeof(float) *
count, data, 0, NULL, NULL);
```

Ak všetky predchádzajúce operácie zbehli bez problémov, môžeme našej kernel funkcii predať argumenty.

```
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
err |= clSetKernelArg(kernel, 2, sizeof(unsigned int), &count);
```

Opať raz sú výsledky priradzované do pomocnej premennej **err** na overenie výsledku volania funkcií. Keď už máme argumenty pre kernel predané, môžeme ho konečne vykonať.

```
err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &global, &local,
0, NULL, NULL);
```

Pri tejto konfigurácii funkcie, dôjde k vykonaniu kernel funkcie na všetkých dostupných výpočtových jednotkách procesora OpenCL zariadenia.

Samozrejme, tak ako aj pri jazyku CUDA, je nutné po skončení operácií s kernel funkciou, dealokovať všetku explicitne alokovanú pamäť. V OpenCL na to slúžia **release** funkcie.

```
clReleaseMemObject(input);  
clReleaseMemObject(output);  
clReleaseProgram(program);  
clReleaseKernel(kernel);  
clReleaseCommandQueue(commands);  
clReleaseContext(context);
```

Ak všetky príkazy zbehnú bez chybových stavov, OpenCL aplikácia je pripravená na použitie na kompatibilných zariadeniach. Samozrejme, v prípade náročnejších programov, ktoré nerátajú len mocniny čísel, treba počítat' s narastajúcim počtom nutných operácií na spustenie programu.

Náročnosť OpenCL kódu je na prvý pohľad oveľa väčšia, ako náročnosť kódu jazyka CUDA. OpenCL však oproti CUDA musí zvládať špecifiká rozličného hardvéru, vysporiadať sa s obmedzeniami ktoré kladú tie ktoré architektúry zariadení. CUDA, na rozdiel od toho, je vyvíjaná hardvérovou spoločnosťou a tak je tento jazyk dôkladne pripravený na danú architektúru. Samozrejme, na iných zariadeniach sa stáva nepoužiteľným.

Z toho sa dá logicky dôjsť k záveru, že vývoj CUDA aplikácií kladie menšie požiadavky na napísanie výkonného kódu. Teda vývoj aplikácii pre túto architektúru môže v blízkej dobe napredovať rýchlejšie, ako v jazyku OpenCL. No má to však aj isté výhody. CUDA sa môže stať odrazovým mostíkom pre vývojárov, ktorý chcú začať tvoriť GPGPU programy. Relatívna jednoduchosť a menšia náročnosť na ošetrovanie rôznych podmienok, vytvárajú ideálne prostredie na vývoj testovacích aplikácií. A po odladení a zefektívnení kódu, zostáva už len malý krok k prepísaniu CUDA aplikácie do multiplatformovej aplikácie OpenCL.

Viac sa tejto problematike budem venovať v kapitole OpenCL verzus CUDA.

OpenCL verzus CUDA

Nakoľko sú na trhu momentálne dostupné dve masívne platformy, poskytujúce vysokú funkcionálnosť na poli GP GPU, naskytá sa otázka, ktorá platforma je výhodnejšia. Táto otázka je však veľmi subjektívna a tak isto je aj odpoveď na ňu. Nakoľko je OpenCL štandard uznávaný viacerými výrobcami môže sa očakávať, že prenositeľnosť kódu bude vysoká. Je to síce výhoda čo sa týka ľahkého šírenia OpenCL aplikácii, na druhej strane však treba platiť cenu v podobe náročnosti kódu a ošetrovania kódu, na kompatibilitu s rozličnými zariadeniami. CUDA na druhej strane neponúka kompatibilitu s inými zariadeniami, nakoľko je to však produkt výrobcu grafických čipov, dá sa očakávať čoraz väčšia integrácia CUDA funkcionálnosť. Či už do kódu alebo priamo v architektúre.

Hlavné rozdiely medzi CUDA a OpenCL sa pokúsim zhrnúť v tomto krátkom zozname.

- OpenCL má značné obmedzenia čo sa týka kopírovania údajov v rámci pamätí dvoch zariadení. To môže spôsobovať problémy pri práci s grafickými kartami, ktoré využívajú dva nezávislé grafické procesory. Ide tu najmä o zapojenie grafických kariet v režime CrossFire X (pre AMD karty) alebo režime SLI (pre NVIDIA karty).
- Synchronizácia procesov v CUDA nie až tak flexibilná ako pri OpenCL. OpenCL umožňuje, pozastaviť akúkoľvek operáciu aby počkala na akúkoľvek inú, zoradenú operáciu. V CUDA 3.2 je už toto obmedzenie odstránené, no spätná kompatibilita so staršími zariadeniami nie je možná.
- CUDA v súčasnosti poskytuje kvalitnejšie prostredie na tvorbu GP GPU programov. Má vlastný kompilátor, debbuger. CUDA svojou syntaxou je veľmi blízka programátorom v jazyku C, zatiaľ čo OpenCL využíva veľa nových kľúčových slov a funkcií.
- CUDA umožňuje využívanie funkcionálnosť jazyka C++, napríklad šablón, v GPU kóde.
- OpenCL lepšie zvláda prácu s klasickými smerníkmi, využívanými v CPU programoch.
- OpenCL umožňuje zápis do textúr, čo CUDA v momentálnej verzii nedokáže.

Nakoľko však v rýchlosti kódu napísaného v CUDA a v OpenCL nie je veľký rozdiel, môžeme povedať, že hlavné rozhodnutie o voľbe správneho prostredia zostáva na programátorovi. Ak by chcel však primárne program, s vysokou rýchlosťou na špecifikom zariadení, siahne radšej po CUDA. OpenCL však umožňuje rovnaký program využívať na rozličnom hardvéri.

Súčasné GPGPU projekty

V súčasnej dobe sa najväčšej pozornosti dostalo GPGPU projektom, ktoré participujú v projektoch distribuovaného spracovania dát. Rozšírenosť výkonných grafických kariet medzi počítačovými entuziastami umožňuje vybudovať obrovské siete počítačov, ktoré spolu ponúkajú obrovský výpočtový výkon. Za zmienku určite stoja projekty univerzity v Berkeley, ktorá stojí za projektom BOINC. Samozrejme, projekty distribuovaného spracovania dát nie sú jedinou oblasťou, v ktorej sa naplno využíva potenciál grafických procesorov. Za zmienku určite stoja simulačné programy, ktoré využívajú paralelné výpočty na simuláciu procesov a javov, ktoré by si pri skutočnom meraní vyžiadali obrovské náklady. Jedná sa o rôzne programy určené na simuláciu dynamiky tekutín, predpovede počasia a podobne. Často sa používajú aj na simulovanie chemických procesov, ktorých sledovanie alebo pozorovanie v reálnom svete je buď veľmi drahé, alebo dokonca nebezpečné. Jedná sa napríklad o simulácie rozkladu prácich prostriedkov vo voľnej prírode.

BOINC

Projekt BOINC predstavuje otvorenú infraštruktúru pre distribuované výpočty. Názov pochádza zo skratky anglických slov **B**erkeley **O**pen **I**nfrastructure for **N**etwork **C**omputing. Samotný BOINC sa ale na práci nepodieľa, ani nijako nezasahuje do výpočtov. Od začiatku bol stavaný tak, že ponúka zastrešenie vedeckých projektov, ktoré chcú presunúť kalkulácie na počítače dobrovoľníkov. Taktiež sa od počiatkov rátalo, že BOINC bude multiprojektový. To znamená, že pod hlavičkou BOINC môžu existovať rozličné projekty, zamerané na rozdielne vedecké oblasti.

Celá myšlienka projektu BOINC je založená na tom, že väčšina osobných počítačov na svete nie je efektívne, alebo vôbec, využitá. Pritom pracovný čas a výkon počítačov sa dá využívať aj bez toho, aby odliv výpočtovej sily užívateľ pocítil. Klient projektu BOINC je navrhnutý tak, aby k jeho spúšťaniu dochádzalo vtedy, ak sa s počítačom nepracuje. Alebo na želanie užívateľa. BOINC klient je vlastne malá aplikácia, ktorá zodpovedá za sťahovanie, riadenie a odosielanie pracovných úloh. Užívateľ si zvolí projekty do ktorých sa chce pripojiť, klient si stiahne čiastkové procesy a podľa požiadaviek užívateľa, pracuje na ich výpočtoch. Po dokončení výpočtov sú výsledky posielané na server a sťahujú sa nové úlohy. Samotný klient sa pritom nestará o kontrolu správnosti, na to už slúži samotný server, ktorý jednotlivé čiastkové výsledky spája dohromady a následne overuje. Samotný užívateľ je odbremený

od zásahov, väčšiu časť práce zvláda klient sám. Výnimkou sú len automatické aktualizácie klienta, ktoré z bezpečnostných dôvodov nie sú povolené.

Autori projektu BOINC možno ani nepočítali s takým úspechom, ktorý sa dostavil aj vďaka tomu, že užívatelia majú možnosť zapájať sa do skupín. Hodnotenia výkonnosti skupín sa pravidelne uvádzajú v tabuľke podľa poradia. Tým sa docielilo, že veľa skupín sa snaží o čo najlepšie postavenie v tabuľke, teda potenciálny výkon projektu BOINC neustále rastie. V súčasnosti je doň zapojených cez pol milióna počítačov, čím sa dosahuje súhrny výkon **5,549 petaFLOPS**. Čiže $5,549 \times 10^{15}$ operácií s desatinou čiarkou za sekundu. Len pre porovnanie, v súčasnosti najvýkonnejší super počítač na svete, čínsky *Tianhe-I* dosahuje výkon **2,566 petaFLOPS**. Určite stojí za zmienku, že tento super počítač je postavený aj na grafických kartách NVIDIA TESLA.

SETI@home

Medzi najznámejšie projekty zastrešené pod BOINC patrí projekt SETI@home. Samotný BOINC bol pôvodne vyvíjaný ako záštita pre tento projekt, neskôr sa obrátil na všeobecnú podporu podobných projektov. Názov SETI@home (*SETI at home* – *SETI doma*) je skratkou z anglických slov Search for **E**xtra-**T**errestrial **I**ntelligence. Teda *Hľadanie mimozemskej inteligencie*. Projekt SETI@home bol spustený do prevádzky 17. mája v roku 1999. V súčasnosti je druhým najväčším projektom distribuovaného spracovania dát.

Hlavnou náplňou tohto projektu je analýza rádiových signálov zachytených rádiovým teleskopom. Konkrétne ide o teleskop pri meste Areciba, nachádzajúcom sa na ostrove Puerto Rico. Sekundárne údaje zachytené týmto teleskopom sú odosielané serveru SETI@home, ten sa stará o ich delenie na čiastkové úlohy. Tie sú potom posielané na analýzu počítačom zapojeným do tohto projektu. Primárnou úlohou projektu je nájsť rádiové signály z vesmíru, ktorým sa nedá priradiť prirodzený pôvod. Dôraz sa kladie nato, aby zachytené vlny neboli len šumom, ale aby sa dalo skutočné dokázať, že ich pôvod je umelý. Ide hlavne o tieto znaky:

- extrémne výkyvy vo vlnovom spektre.
- klesanie a nárast prenosu, ktoré spadá do Gaussovho rozdelenia.
- trojčatá – ide o tri extrémne výkyvy v poradí.
- pulzujúce signály ktoré predstavujú potenciál úzkopásmového digitálneho prenosu.
- autokorelácia vlnového pásma.

Nakoľko však v súčasnosti zachytávame z vesmíru obrovské množstvo signálov, je táto úloha nesmierne náročná. Aby sa dalo jednoznačne prehlásiť, že zachytený signál má umelý pôvod, je nutné ho podrobiť viacerým testovaniam. Na zachytenom signáli sa prevádzajú simulácie, ktoré ho zosilňujú alebo oslabujú. Pri tomto simulovaní sa hľadajú extrémne hodnoty, ktoré zatiaľ neboli priradené žiadanému úkazu, ktorý by mohol spôsobovať ich deformáciu. Jedná sa napríklad o gravitačnú deformáciu spôsobenú planétami alebo inými telesami.

Aj keď je do projektu SETI@home zapojených množstvo užívateľov a beží už dlhú dobu, k dnešnému dňu sa nepodarilo dokázať, že zachytené signály mali umelý pôvod.

Folding@home

Folding@home je projekt distribuovaného spracovania dát zameraný na výpočet simulácií skladania a rozkladania proteínov a ďalších simulácií molekulárnej dynamiky. Taktiež sa snaží o zlepšenie metód na výpočet daných úloh. Projekt bol spustený 1. októbra 2000 univerzitou v Standforde v USA. V súčasnosti je riadený skupinou Pande Group, za ktorou stojí profesor Vijay Pande. Projekt Folding@home je v súčasnosti najvýkonnejší a najväčší projekt distribuovaného spracovania dát. Dosahuje výkon až 12,4 petaFLOPS, čo je dvakrát viac ako projekt SETI@home a takmer šesťkrát viac ako super počítač *Tiahne-I*. Momentálne je do projektu zapojených takmer pol milióna počítačov a za celú dobu funkčnosti za projektu zúčastnilo cez šesť miliónov počítačov.

Hlavnou náplňou projektu je pochopiť skladanie a rozklad proteínov a chorôb vznikajúcich pri týchto javoch. Presné simulácie umožňujú lepšie pochopiť príčiny vzniku a rozvoj chorôb ako sú rakovina, Alzheimerova choroba, Parkinsonova choroba a mnohé iné. Práve lepšie pochopenie toho, ako sa biologické molekuly skladajú funkčných reťazcov, je veľkým problémom molekulárnej biológie.

Systém práce je obdobný tomu v projektoch BOINC. Aplikácia fungujúca na klient - server architektúre, delenie výpočtov na čiastkové úlohy a ich spätné skladanie serverom. Klient umožňuje aby výpočty bežali ako šetrič obrazovky, teda v čase, keď užívateľ s počítačom nepracuje. Samozrejmosťou projektu je súťažný element, teda motivácia pre účastníkov o dosahovanie čo najlepších výsledkov, čo sa týka objemu spracovaných dát. Klient je po dlhom čase funkčnosti schopný využívať potenciál OpenCL alebo CUDA zariadení. A to aj

v prípadoch zapojenia viacerých zariadení, či už v režime CrossFire alebo NVIDIA SLI. Folding@home umožňuje aj zapojenie herných konzol do výpočtov, konkrétne konzoly PlayStation 3 od spoločnosti SONY. Táto konzola so svojím osem jadrovým procesorom na architektúre CELL taktiež výrazne prispieva do výpočtovej sily celého projektu.

Po dlhoročnej funkčnosti projektu, sa podarilo úspešne nasimulovať už viacero spôsobov skladania proteínov. Taktiež sa stále zlepšuje presnosť výpočtov a taktiež časová variabilita skladania proteínov. Práve čas skladania môže výrazne ovplyvniť výsledok skladania, teda či budú molekuly zložené bezchybne, alebo práve s deformáciami, ktoré následne spôsobujú ochorenia.

Iné využitie GPGPU

Okrem projektov distribuovaného spracovania dát, existuje v dnešnej dobe množstvo ďalších aplikácií ktoré dokážu využívať potenciál grafických procesorov. Stále však majú rovnaký základ, a to náročnosť výpočtu a možnosť paralelnej exekúcie. Ako sa už dá dedukovať z predchádzajúcich kapitol, GPGPU aplikácie naplno ukazujú svoj potenciál v simulačných výpočtoch.

V tomto odvetví sú veľmi zaujímavé projekty, ktoré sa usilujú o simuláciu zraku. Nejde tu ale o žiadne umelé videnie pre ľudí, skôr o schopnosť počítačov extrahovať informácie z digitálne uložených obrázkov. Inšpirácie pre tieto projekty pochádza z vesmírneho výskumu, a to najmä z projektov planetárnych sond. Schopnosť robota samostatne rozoznať zachytené obrazce, môže výrazne ovplyvniť spôsoby výskumu iných planét alebo mesiacov. Latencia komunikácie medzi sondou a riadiacim strediskom spomaľuje výskum, a preto istá miera samostatného usudzovania sond môže zlepšiť výsledky výskumu.

Čo sa týka sféry osobných počítačov a komerčného softvéru, najčastejšie sa GPU computing využíva pri video a audio programoch. Pri video dekodovaní sa okrem toho dá veľmi efektívne využiť aj pôvodný účel grafických kariet. Teda okrem využitia grafického procesora na dekodovanie video súborov, sa vo veľkej miere snažia softvéroví vývojári aj o využitie grafických funkcionalít procesorov. Tu sa hlavne jedná o využite techník ako vyhladzovanie hrán, používanie svetelných filtrov, potlačanie šumu a iné funkcionality. Výsledkom tak nie

sú len rýchlejšie dekódované videá, ale zároveň aj vylepšené po obrazovej stránke. Pritom sa na nich odrážajú aj vlastnosti jednotlivých architektúr grafických kariet a ich primárnych oblastí. Grafické karty od spoločnosti ATI sa pri dekódovaní videa zameriavajú skôr na vernosť osvetlenia a hĺbku obrazu. Naopak grafické karty od NVIDIA ponúkajú ostrejší obraz, čo však na druhej strane niekedy deformuje vernosť textúr.

Výpočtová sila grafických kariet sa pomaly pretláča aj do ďalších programov, ktoré sú často využívané aj bežnými užívateľmi. Určite stojí za zmienku, že čoraz viac výrobcov antivírusových programov, začína zavádzať algoritmy, ktoré majú urýchliť skenovanie systému a zrýchliť tak priebeh testovania. Tu sa však naráža na ďalší problém. Tým je práve obmedzenie rýchlosti ostatnými hardvérovými súčasťami počítača. Mechanické disky majú obmedzenú rýchlosť čítania a zapisovania, takže aj keď dokážeme hranične využiť potenciál grafických a centrálnych procesorov, stále však tento výkon závisí od najpomalšieho zariadenia, ktoré je v danej konfigurácii zapojené.

Záver

GPU computing ponúka v súčasnosti unikátny spôsob, ako dosiahnuť na osobnom počítači výkon super počítača. Pritom na konečného používateľa sú kladené len minimálne nároky. Okrem zapnutia počítača a spustenia aplikácie, nemusí o paralelných výpočtoch vedieť nič. A pritom mu kvalitná grafická karta spolu s odladeným softvérom ponúknu neskutočné možnosti.

S rastúcim významom tohto segmentu možno očakávať, že výrobcovia grafických procesorov budú čoraz viac ponúkať hardvér primárne určený do tejto sféry. Už teraz je na trhu viacero grafických kariet, určených na vedeckú činnosť, ktoré neponúkajú možnosť výstupu na obrazovku. Stali sa z nich prakticky zásuvné, vysoko paralelné procesory.

Použité zdroje

- [1] SANDERS, J. – KANDROT, E. 2011. *CUDA by example*. Boston: Pearson Education , Inc.. 2011. 307 s. ISBN 0-13-138768-5
- [2] KIRK, DAVID B. –WEN –MEI HWU. 2010. *Programming massively parallel processors*, Burlington: Morgan Kaufman Publishers. 2010. 270 s. ISBN 978-0-12-381472-2
- [3] NVIDIA Corporation. 2009. *CUDA C Best Practices Guide*. [exe]. Santa Clara: NVIDIA Corporation
- [4] NVIDIA Corporation. 2009. *CUDA C Programming Guide*. [exe]. Santa Clara: NVIDIA Corporation
- [5] Advanced Micro Devices, Inc. 2009. *ATI Stream SDK OpenCL Programming Guide* [exe]. Sunnyvale: Advanced Micro Devices, Inc.
- [6] kolektív autorov. *AMD Fire Stream* [online]. [dostupné 30.03. 2011] Dostupné na internete: http://en.wikipedia.org/wiki/AMD_FireStream
- [7] kolektív autorov. *CUDA* [online]. [dostupné 30.03. 2011] Dostupné na internete: <http://en.wikipedia.org/wiki/CUDA>
- [8] kolektív autorov. *OpenCL* [online]. [dostupné 30.03. 2011] Dostupné na internete: <http://en.wikipedia.org/wiki/OpenCL>
- [9] kolektív autorov. *GPGPU* [online]. [dostupné 30.03. 2011] Dostupné na internete: <http://en.wikipedia.org/wiki/GPGPU>