Katedra počítačov a informatiky

Fakulta elektrotechniky a informatiky

Technická univerzita v Košiciach

Branislav Sobota

Aplikačné programové rozhrania pre GPGPU

Pomocný učebný text

(len pre interné použitie)

2012

Obsah

Tento učebný text je určený ako interný rozširujúci pomocný učebný text pre predmet Systémy virtuálnej reality. Text obsahuje základný opis aplikačného programového vybavenia pre GPGPU s fokusáciou na CUDA a OpenCL

Text neprešiel celkovou gramatickou, štylistickou a formátovacou úpravou

1 Úvod

GPGPU aplikačné rozhrania a jazyky predstavujú middleware medzi aplikáciou a grafickým štandardom OpenGL alebo Direct3D. Ich hlavnou úlohou je uľahčenie programovania pomocou transformácie syntaxe jazyka C alebo C++ do kódu shader jazyka alebo kódu niektorého z grafických štandardov. Táto transformácia sa vykonáva pomocou CPU procesora. GPGPU jazyk *Brook* [4] je vyvíjaný na Stanfordskej Univerzite a je voľne dostupný. Avšak obe vedúce spoločnosti v tejto oblasti nVIDIA aj ATI predstavili vlastné propiertárne riešenia. Ďalším bojovníkom v tejto oblasti je aj Microsoft s technológiou *DirectCompute* (napr na `http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=16995`,). Starším a najviac používaným je metajazyk pre programovanie GPU spoločnosti nVIDIA s názvom *CUDA* (`http://www.nVIDIA.com/object/cuda_home_new.html`). CUDA [18] (Compute Unified Device Architecture, predstavuje proprietárnu možnosť programovať GPU resp. unified shader jednotky pomocou sady funkcií a podporných knižníc [24]. CUDA predstavuje C++ prekladač, ktorý navyše obsahuje viacero rozšírení jazyka C++, ktoré dovoľujú písať kód v jazyku C++, s ktorým dokáže priamo pracovať GPU. Tieto rozšírenia sú podporované softvérovými knižnicami a špeciálnym ovládačom CUDA, ktorý v GPU predstavuje operačný systém s aplikáciami ako matematický koprocesor. CUDA je produkt firmy nVIDIA a ponúka celú sadu nástrojov na odladenie kódu, t.j. prekladač, debugger a tester. Medzi jeho nevýhody je možné zaradiť problematickú podporu riadenia výpočtov na viacerých grafických adaptéroch, paralelne zapojených v jednom počítačovom systéme pomocou technológie SLi [23]..

Na zatiaľ chronologicky poslednom mieste je OpenCL (Open Computing Language). OpenCL (http://www.khronos.org/opencl, [8]) predstavuje prvý pokus o zjednotenie prostredí pre vývoj aplikácií GPGPU aj keď vo svojej podstate je určený ako viacplatformové heterogénne riešenie pre písanie aplikácií využívajúcich paralelný výpočtový proces. OpenCL poskytuje jednotné programovacie prostredie pre vývojárov softvéru, písať účinné, prenosné programové vybavenie pre vysokovýkonné výpočtové systémy, stolné počítače a mobilné zariadenia s použitím rôznych kombinácií viacjadrových procesorov, grafických procesorov a príp. ďalších paralelných architektúr. OpenCL predstavuje otvorený priemyselný štandard vyvinutý firmou Apple Inc. a spravovaný firmou Khronos Group. Prihlásili sa k nemu všetci väčší výrobcovia hardvéru vrátane nVidie a ATi. Vo svojej podstate možnosti využitia GPGPU v paralelnom prostredí spočívajú v masívnom využití dátovo – paralelného spracovania dát. Štandard OpenCL sa skladá z API rozhrania určeného na riadenie paralelných výpočtov a programovacieho jazyka, ktorý je špeciálne určený na zadávanie týchto výpočtov. Tiež sa využívajú vlastnosti programovacieho jazyka C99. API tiež zosúlaďuje paralelné výpočty úloh a dát s rôznymi typmi mikroprocesorov, grafickými procesormi a tiež spolupracuje s OpenGL, OpenGL ES a ďalšími grafickými API. V paralelnom prostredí môžu nastať dve možnosti využitia pracovnej stanice zapojenej do systému:

- Stanica pracuje samostatne na zadanej úlohe od nadradenej stanici – v tomto prípade sa GPU procesor využíva na nezávislé spracovanie konkrétnej úlohy a výsledok po jej úspešnom dokončení posiela nadradenej stanici
- Stanica pracuje spoločne na zadanej úlohe od nadradenej stanici – v takomto prípade je GPU procesor súčasťou širšieho riešiteľského rámca a na rozdiel od nezávislého spracúvania konkrétnej úlohy vzájomne komunikuje (synchronizácia, posielanie správ a čiastkových výsledkov) s okolitými stanicami

Samozrejme, v súčasnosti je najvhodnejším technologickým a zároveň aj softvérovým riešením paralelných úloh práve riešenie od samotných výrobcov grafického hardvéru, ktoré je ale prioritne orientované pre riešenie grafických úloh.

2 Aplikačné rozhranie API CUDA

CUDA C je rozšírením C, ktoré nVIDIA vyvinula za účelom kompilovania programov obsahujúcich kódy CUDA kernelov. Tým vznikol jazyk pre písanie kódu pre GPU, nVIDIA taktiež uviedla špecializovaný hardvér na spúšťanie masívnej výpočtovej sily CUDA architektúry. CUDA C je dostupná v balíkoch CUDA Toolkit vo verzii 4.0.

Paralelný programovací model CUDA bol navrhnutý aby prekonával mnoho výziev paralelného programovania. V základoch je postavený na troch abstrakciách:

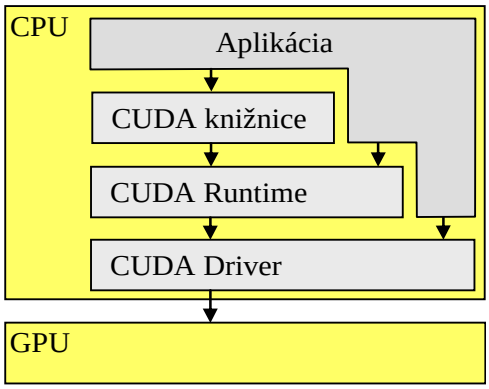
- hierarchia skupín vláken
- zdieľaná pamäť
- synchronizácia vláken

Tieto abstrakcie sú ponúkané vývojárom ako malá sada jazykových rozšírení. Poskytujú viacero druhov paralelizmu, pritom však vyžadujú explicitne rozdelenie problémového priestoru do podproblémov, ktoré môžu byť vyriešené paralelne nezávisle. Táto dekompozícia ponúka škálovateľnosť CUDA architektúry do grafických kariet rozličných počtov procesorov a rozličnej pamäťovej kapacity. CUDA poskytuje dve API k tvorbe programov:

- vysoko-úrovňové *CUDA Runtime API*
- nízko-úrovňové *CUDA Driver API*

Keďže je CUDA Runtime API implementované nad CUDA Driver API, je každé volanie funkcie z Runtime rozložené do základnejších inštrukcií spravovaných Driver API (Obr. 1). Ich vzájomné použitie sa vylučuje, teda programátor si musí vybrať, ktoré API zvolí. Komplexnejšia a zložitejšia je práca s Driver API, pretože potrebuje hlbšie pochopenie samotnej architektúry GPU, ale zato poskytuje väčšiu flexibilitu. Obidve API sú schopné komunikovať s OpenGL alebo Direct3D, a príkladov ich interakcií je mnoho. Ako prídavné softvérové vrstvy sú ponúkané ďalšie množiny knižníc, napr.:

- *CUBLAS* - množina prvkov pre kalkulácie lineárnej algebry
- *CUSP* – knižnica pre zriedkavú lineárnu algebru a grafové výpočty
- *CUFFT* - knižnica schopná kalkulácií Furierových transformácií
- *Thrust* – knižnica paralelných algoritmov s rozhraním podobným C++ STL (Standard Template Library)



Obr. 1 CUDA architektúra

Pri návrhu CUDA architektúry sa firma nVIDIA rozhodla vo svojich dokumentáciách využívať nový slovník špecificky pre danú technológiu. Zdanlivo známe termíny majú v tomto ponímaní rozdielny význam, a tak je potrebné uviesť niektoré pojmy, ktoré budú používané v nasledovnom.

- *thread (vlákno)* - je základným dátovým elementom, ktorý sa spracováva. Na rozdiel od vláken CPU sú CUDA vlákna podstatne odľahčené, čo znamená, že výmena obsahov dvoch vláken nie je nákladnou operáciou.
- *warp (osnova)* - skupina 32 vláken, čo je minimálna veľkosť dát v SIMD (Single Instruction, Multiple Data) spracovávaná CUDA multiprocesorom
- *block (blok)* - môže obsahovať od 64 do 512 vláken, a slúži k efektívnejšiemu používaniu, aby v prípade potreby nebolo nutné manipulovať priamo s osnovami.
- *grid (matica, mriežka)* – obsahuje zoskupené bloky. Výhodou takéhoto zoskupovania je, že bloky sú spracovávané súčasne. Množstvo blokov v mriežke umožňuje abstrahovať od obmedzení a aplikovať jadro na rozsiahlu množinu vláken v jedinom volaní. V prípade ak má hardvér málo dostupných zdrojov, spúšťa bloky sekvenčne, a naopak, ak má zdrojov dostatok, vykoná ich paralelne.
- *host (hostiteľský počítač)* - označenie stroja CPU
- *device (zariadenie)* - označenie zariadenia GPU

2.1 Vývojový model CUDA

Aplikácia pozostáva z hlavného programu, ktorý sa vykonáva na hostiteľskom (host) CPU a CUDA program, ktorý sa vykonáva na GPU akcelerátora. Hostiteľský program volá výpočtové jadro (kernel, prípadne aj viac jadier), ktorý sa následne vykonáva na GPU. Kernel je vykonáva paralelné operácie pomocou vlákien v GPU. Výpočtový kernel je napísaný buď v CUDA C, alebo PTX.

Optimalizovanie CUDA programu pozostáva z hľadania optimálneho vyváženia medzi počtom blokov a ich veľkosťou. Na maskovanie latentnosti pamäťových operácií je výhodnejšie mať bloky obsahujúce viac vláken, ale na druhej strane je redukovaný počet dostupných registrov na vlákno. nVIDIA odporúča používanie blokov o veľkosti od 128 do 256 vláken, čím je ponúkaný najlepší kompromis medzi maskovaním latentnosti a množstvom potrebných registrov pre jadrá.

2.2 Kernel (jadro)

CUDA C rozširuje jazyk C tým, že dovoľuje programátorovi definovať C funkcie, zvané Kernel.

Kernel je definovaný pomocou deklarácie *__global__*. Samotné volanie kernela, kde sa špecifikuje počet CUDA vlákien zapojených do výpočtu, prebieha pomocou *<<< N , M>>>* syntaxe. Kde *N* označuje počet blokov a *M* počet vlákien v jednom bloku. Každému vláknu, ktoré je zapojené do vykonávania kernela je priradená vstavaná špecifická premenná *threadIdx*, ktorá je vlastne identifikátorom (ID) konkrétneho vlákna [17].

2.3 Hierarchia vlákien

Premenná *threadIdx* je vlastne trojzložkový vektor, takže vlákna môžu byť označené pomocou jedno, dvoj, alebo trojrozmerného bloku vlákien. Tento prístup umožňuje volanie výpočtov naprieč vektormi, maticami, alebo kapacitami.

Index vlákna a jeho ID sa riadia jednoduchými pravidlami:

- V jednorozmernom bloku sú tieto hodnoty zhodné
- V dvojrozmernom bloku veľkosti *(Dx, Dy)* je *ID* vlákna s indexom *(x, y) = (x + y Dx)*
- V trojrozmernom bloku veľkosti *(Dx, Dy, Dz)* je *ID* vlákna s indexom *(x, y, z) = (x + y Dx + z Dx Dy)*

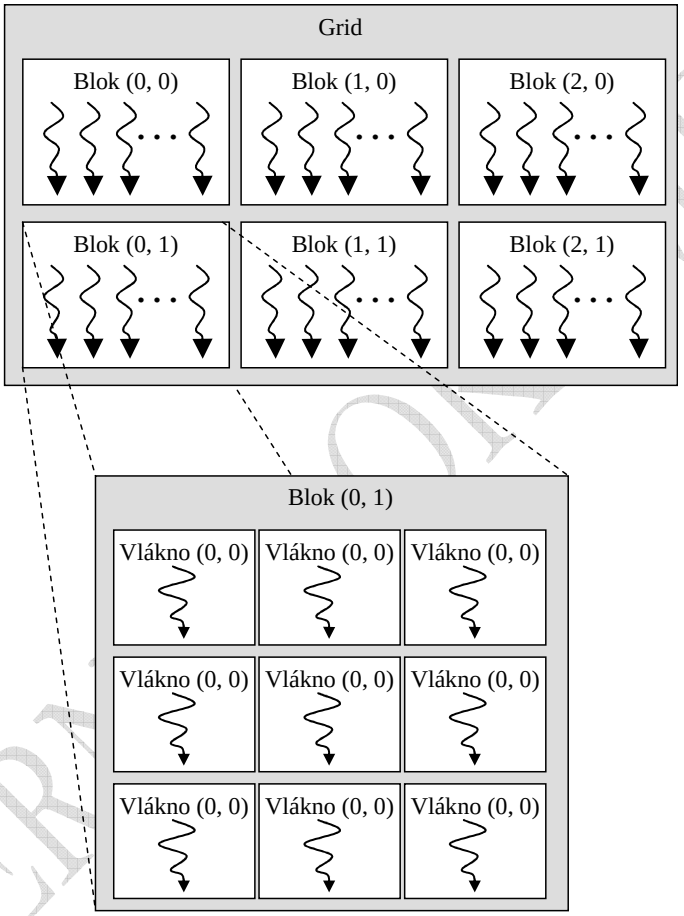
Maximálny počet vlákien v bloku ohraničený a to z dôvodu, že všetky vlákna v bloku musia sídliť v rovnakom procesorovom jadre a zdieľať určitú minimálnu časť pamäťových prostriedkov. Avšak kernel

môže byť vykonávaný na viacerých rovnako veľkých blokoch vlákien, takže celkový počet dostupných vlákien je rovný počtu vlákien v bloku, krát počet blokov.

Bloky sú usporiadané do jedno, alebo dvojrozmerných gridov blokov vlákien tak, ako je znázornené na obrázku (Obr. 2). Počet blokov v gride je obvykle udávaný rozmermi údajov ktoré sa spracovávajú, alebo počtom procesorov v systéme, ktoré však môže výrazne prevýšiť.

Počet vlákien v bloku a počet blokov v gride špecifikovaných v <<<…>>> syntaxe môže byť typu int, alebo dim3. Dvojrozmerný grid blokov môže byť špecifikovaný nasledovne:

```
// Volanie Kernela s jedným blokom ( N * N * 1 ) vlákien
int numBlocks = 1;
dim3 threadsPerBlock(N, N);
MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```



Obr. 2 Grid blokov vlákien

Každý blok v gride môže byť identifikovaný pomocou jedno, alebo dvojrozmerného indexu dostupného v jadre pomocou vstavanej premennej *blockIdx*. Rozmery bloku vlákien sú dostupné prostredníctvom volania premennej *blockDim*. Najbežnejší rozmer bloku vlákien je 16 x 16 (256 vlákien).

Bloky vlákien je nutné vykonávať samostatne. Musí byť možné vykonávať ich v rôznom poradí, paralelne, alebo sériovo. Potreba tejto nezávislosti umožňuje blokom vlákien aby boli plánovateľné v rôznom poradí na rôznom množstve jadier (Obr. 5).

Vlákna v bloku môžu spolupracovať zdieľaním údajov prostredníctvom zdieľanej pamäte a pomocou synchronizácie ich vykonávania pre koordináciu prístupu do pamäte. Synchronizácia sa môže volať v jadre pomocou vstavanej funkcie *__syncthreads()*. Táto funkcia slúži ako bariéra, pri ktorej musia všetky vlákna v bloku počkať kým je ktorejkoľvek povolené pokračovať. Pre efektívnu spoluprácu je zdieľaná pamäť navrhnutá aby pracovala s nízkym oneskorením pre každé procesorové jadro (podobne ako L1 cache) [17].

2.4 Hierarchia pamätí

CUDA vlákna môžu počas ich vykonávania pristupovať k dátam z viacerých pamäťových miest (Obr. 3). Každé vlákno má privátnu lokálnu pamäť. Každý blok vlákien má zdieľanú pamäť, viditeľnú pre všetky vlákna rovnakého bloku a s rovnakou životnosťou ako samotný blok. Všetky vlákna majú prístup k rovnakej globálnej pamäti.

Sú tu taktiež dve dodatočné pamäťové miesta typu „read-only" dostupné všetkými vláknami: konštantné a textúrové pamäťové miesta. Globálne, konštantné a textúrové pamäťové miesta sú optimalizované pre rôzne pamäťové využitia. Textúrová pamäť poskytuje rôzne adresné modely, či filtrovanie dát pre špecifické dátové formáty. Globálne, konštantné a textúrové pamäťové miesta sú dostupné nepretržite počas behu jadra v rovnakej aplikácií [17].

2.4.1 Pamäť zariadenia (Device pamäť)

Tak ako je popísané v kapitole 2.5, CUDA programovací model predpokladá systém pozostávajúci z host-u (hostiteľa) a device (zariadenia), každý s ich vlastnou oddelenou pamäťou. Jadrá môžu operovať iba v rámci pamäti zariadenia (device memory), takže runtime poskytuje funkcie pre alokáciu, dealokáciu a kopírovanie dát medzi pamäťou hostiteľa (host memory) a pamäťou zariadenia (device memory).

Pamäť zariadenia môže byť alokovaná ako lineárna pamäť, alebo ako CUDA polia.

CUDA polia sú vhodné pre prácu s textúrami. Lineárna pamäť existuje na zariadení v 32 bitovom adresnom mieste pre zariadenia výpočtovej schopnosti 1.x a 40 bitovom adresnom mieste pre zariadenia s výpočtovou schopnosťou 2.x, takže oddelene alokované entity môžu odkazovať na iné pomocou smerníkov, napríklad v binárnom strome.

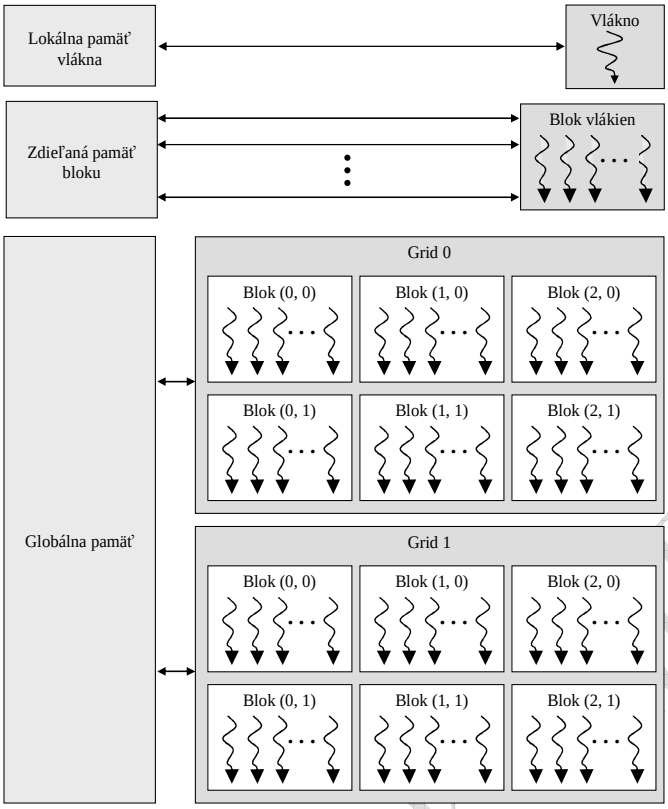
Lineárna pamäť sa alokuje použitím *cudaMalloc()* a uvoľňuje sa použitím *cudaFree()*. Premiestňovanie dát medzi host memory a device memory sa uskutočňuje použitím *cudaMemcpy()*.

2.4.2 Zdieľaná pamäť

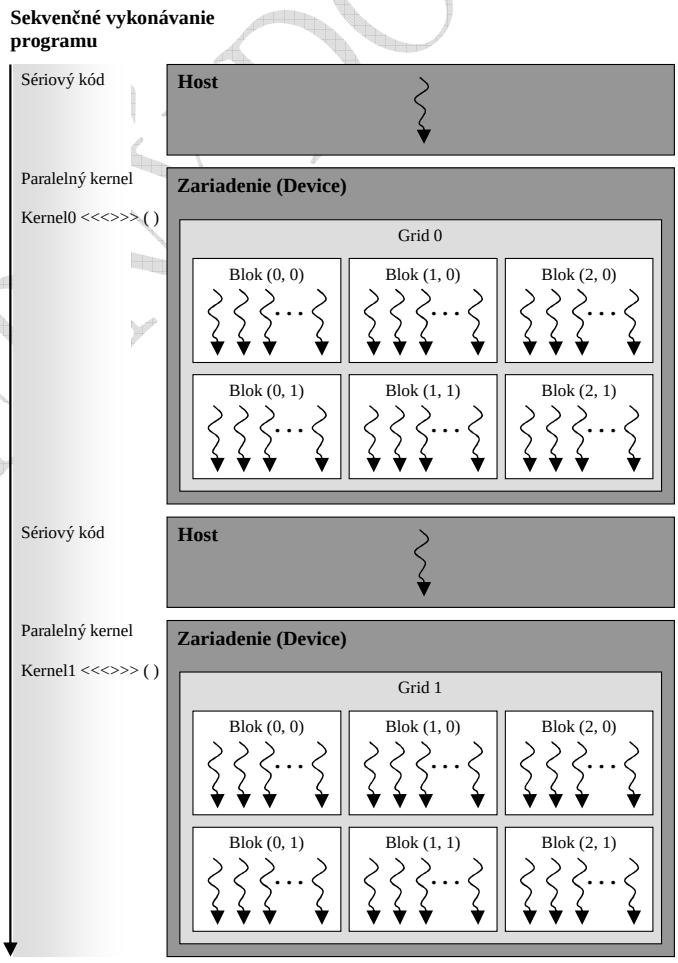
Alokácia zdieľanej pamäte prebieha pomocou identifikátora *__shared__*. Zdieľaná pamäť je omnoho rýchlejšia ako globálna pamäť. Každá možnosť nahradenia použitia globálnej pamäti zdieľanou pamäťou by mala byť využitá, pretože môže viesť k markantným časovým úsporám pri výpočtoch [17].

2.5 Heterogénne programovanie

CUDA programovací model predpokladá spúšťanie CUDA vlákien na fyzicky oddelenom zariadení (device), ktoré operuje ako koprocesor k hosťovskému zariadeniu (host), na ktorom beží C program (Obr. 4). V tomto prípade sa kernel spúšťa na GPU a zvyšok C programu sa vykonáva na CPU. CUDA programovací model rovnako predpokladá, že obe host, aj device spravujú ich vlastné oddelené pamäťové miesta v DRAM – host / device memory (pamäť) [17].



Obr. 3 Hierarchia pamätí (CUDA)



Obr. 4 Sekvenčné vykonávanie programu (CUDA)

2.6 Stupňovateľný programovací model

Príchod viacjadrových CPU a mnohojadrových GPU znamenal, že hlavné procesorové čipy sa stali paralelnými systémami. A čo viac, ich paralelizmus napreduje v mierke Moorovho zákona [11].

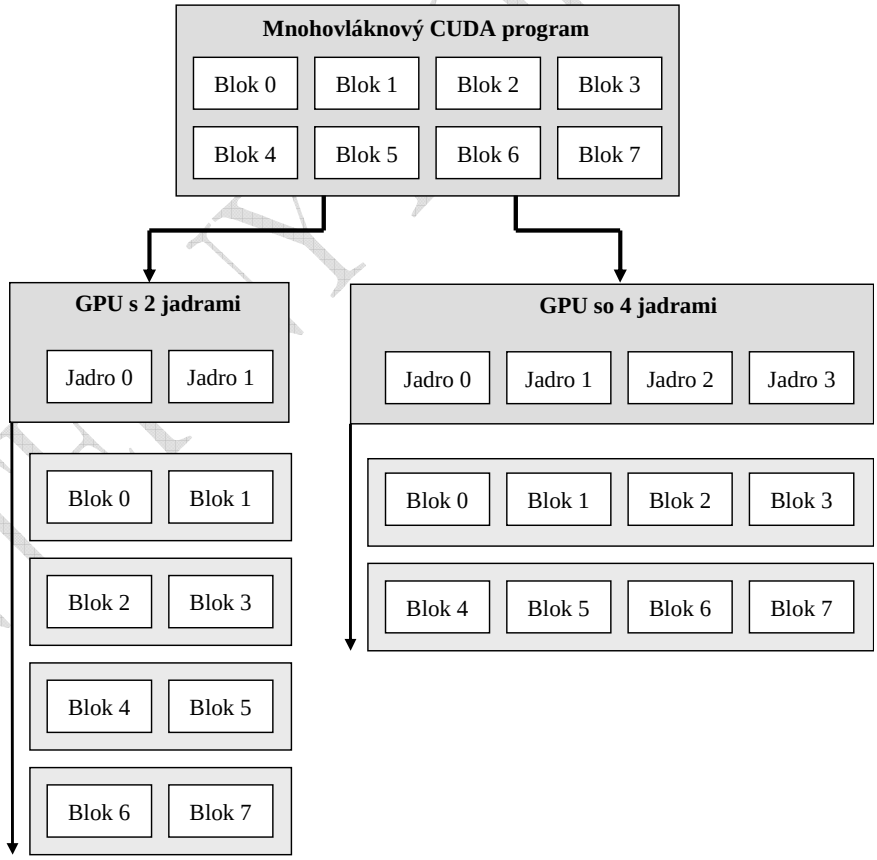
Paralelný programovací model CUDA je navrhnutý tak, aby umožnil masívne paralelné výpočty s využitím maximálneho množstva jadier GPU bez toho, aby nútil vývojárov radikálne meniť ich zvyklosti.

Jadro obsahuje 3 kľúčové abstrakcie:

- hierarchia skupín vlákien,
- zdieľané pamäte,
- barierová synchronizácia,

ktoré sú programátorovi dané ako minimálny balík jazykových rozšírení.

Tieto abstrakcie poskytujú jemnozrnný paralelizmus a vláknový paralelizmus, vnorený do hrubozrnného dátového paralelizmu a paralelizmu úloh. Tie vedú programátora k deleniu problémov na menšie pod problémy, ktoré môžu byť riešené individuálne, rozparalelnením na bloky vlákien a každý pod problém na menšie časti, ktoré môžu byť vyriešené hromadne pomocou všetkých vlákien v bloku. Táto dekompozícia chráni jazykovú expresívnosť tým, že umožňuje vláknam, aby spolupracovali pri riešení každého pod problému a v rovnakom čase umožňuje automatické stupňovanie. V skutočnosti môže byť každý blok vlákien plánovaný na ktoromkoľvek dostupnom procesorovom jadre, v rozličnom usporiadaní - súbežnom, alebo sekvenčnom, takže kompilovaný CUDA program môže byť vykonaný na akomkoľvek množstve procesorových jadier (Obr. 5) [17].



Obr. 5 Stupňovateľný programovací model (CUDA C)

2.7 Viacero zariadení

Host systém môže vykonávať obsluhu viacerých device. Tieto device môžu byť vymenované, ich vlastnosti môžu byť volané a jeden z nich môže byť zvolený pre vykonávanie kernelov.

Viacero hostovských vlákien môže vykonať device kód na rovnakom zariadení, ale podľa dizajnu, hostovské vlákno môže vykonať device kód iba na jednom zariadení v danom čase. Ako dôsledok, viacero hostovských vlákien je potrebných pre vykonávanie device kódu na viacerých zariadeniach.

Nasledujúci kód volá všetky zariadenia v systéme a získava ich vlastnosti. Taktiež zisťuje počet CUDA zariadení:

```
int PocetZariadeni;
cudaGetDeviceCount(&PocetZariadeni);
int device;
for (device = 0; device < PocetZariadeni; ++device) {
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, device);
if (dev == 0) {if (deviceProp.major == 9999 && deviceProp.minor == 9999)
printf("Nieje pripojene ziadne CUDA zariadenie.\n");
else if (deviceCount == 1)
printf("Je pripojene 1 CUDA zariadenie.\n");
else
printf("Je pripojenych %d CUDA zariadeni\n",
deviceCount);
} }
```

V základnom nastavení je device asociované k host vláknu implicitne označené ako device 0. Každé ďalšie zariadenie (device) môže byť označené pomocou volania *cudaSetDevice()*. Po označení zariadenia, je jedno či implicitne, alebo explicitne označeného, každé ďalšie volanie *cudaSetDevice()* bude neúspešné, pokiaľ nebude zavolané *cudaThreadExit()*. *cudaThreadExit()* čistí všetky zdroje spojené s behom programu volané hostovským vláknom [17].

3 Aplikačné rozhranie API OpenCL

OpenCL (Open Computing Language) je otvorený štandard pre všeobecné použitie paralelného programovania na CPU, GPU a ďalších procesoroch, čo vývojárom softvéru prináša efektívny prístup k výkonu týchto heterogénnych procesorových platforiem.

OpenCL podporuje širokú škálu aplikácií, od vstavaného a spotrebiteľského softvéru pre HPC (High Performance Computing) riešenia. Vytvorením účinného, nízkoúrovňového programovacieho rozhrania, bude OpenCL tvoriť základnú vrstvu ekosystému paralelných výpočtov pre platformovo-nezávislé nástroje, middleware a aplikácie.

OpenCL sa skladá z API (Application Programming Interface) pre koordináciu paralelného výpočtu naprieč heterogénnymi procesormi, a multiplatformového programovacieho jazyka s dobre špecifikovaným výpočtovým prostredím **[8]**. Štandard OpenCL:

- Podporuje údajovo, aj problémovo orientované paralelné programovacie modely.
- Využíva podmnožinu C99 ISO s rozšíreniami pre paralelizmus.
- Definuje konzistentné numerické požiadavky založené na IEEE 754.
- Definuje nastavenia profilu pre príručné a vstavané zariadenia.
- Efektívne spolupracuje s OpenGL, OpenGL ES, a inými grafickými API.

OpenCL spravuje technologické konzorcium Khronos Group. Khronos Group je nezisková organizácia, ktorá je financovaná od svojich členov. Bola založená v roku 2000 skupinou mediálne orientovaných spoločností 3Dlabs, ATI, Discreet, Evans & Sutherland, Intel, nVIDIA, SGI a Sun Microsystems. Tieto spoločnosti sa spojili za účelom vytvárania API otvorených štandardov pre paralelné výpočty, grafiku a dynamické médiá spustiteľné na širokej škále platforiem a zariadení **[7]**.

OpenCL a CUDA toho majú mnoho spoločného, no v mnohých veciach sa odlišujú. Základné rozdiely v názvoch jednotlivých druhov pamätí a ich vnútornej hierarchie sa nachádza v tabuľke (Tab. 1).

CUDA	OpenCL
Globálna pamäť	Globálna pamäť
Zdieľaná pamäť	Lokálna pamäť
Lokálna pamäť	Privátna pamäť
Kernel	Program
Blok	Pracovná skupina
Vlákno	Pracovná jednotka

Tab. 1 Rozdiely v pomenovaní CUDA a OpenCL

3.1 Vývojový model

OpenCL aplikácia pozostáva z host programu a OpenCL programu, ktorý sa vykonáva na výpočtovom zariadení (device). Hostovský program aktivuje výpočtové kernele, ktoré sú obsiahnuté v device programe. Kernele sú vykonávané paralelne na device pre viacero dátových jednotiek prostredníctvom device procesných elementov. OpenCL kernele sú písané v OpenCL C programovacom jazyku. Hostovský program kontroluje device použitím OpenCL API. Grafické operácie sú poskytované prostredníctvom OpenGL.

3.2 Vykonávací model

Keďže sa OpenCL nesústredí iba na GPU, ale taktiež na mnohojadrové CPU, OpenCL rozlišuje dva typy kernelov:

- dátovo paralelné, ktoré sú vhodné pre architektúru GPU,
- úlohovo paralelné, ktoré sú vhodné pre architektúru CPU.

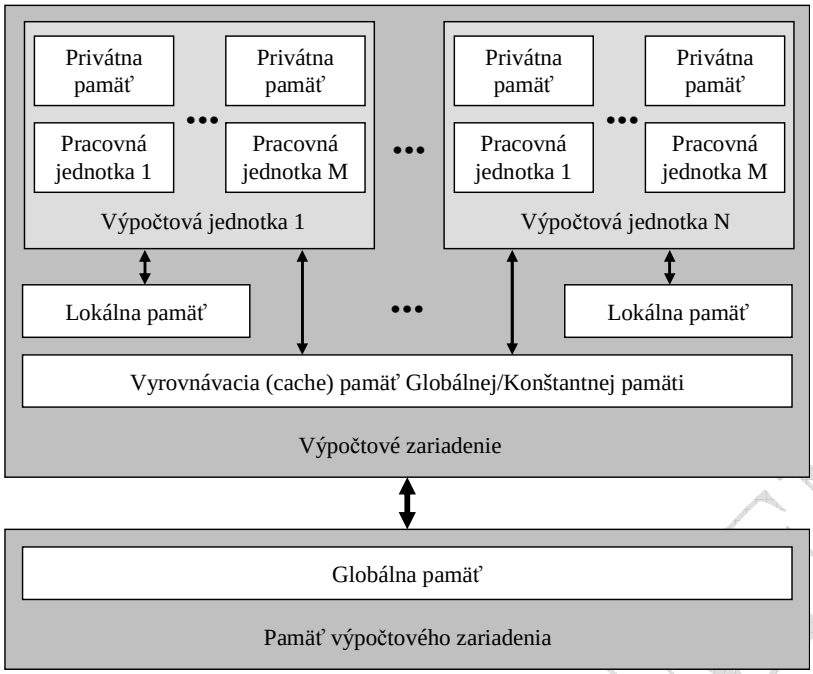
Výpočtový kernel je základná jednotka vykonateľného kódu, ktorý je podobný C funkcií. Vykonávanie kernelov môže prebiehať buď postupne, alebo mimo postupnosti, v závislosti na parametroch, ktoré boli odoslané systému keď zoraďuje kernele pre vykonanie [1].

3.3 Model pamätí

OpenCL pracuje s viacúrovňovým pamäťovým modelom (Obr. 6), radeným od privátnej pamäte, viditeľnej iba pre jednotlivé výpočtové jednotky, až po globálnu pamäť, ktorá je viditeľná pre všetky výpočtové jednotky na zariadení (device).

OpenCL definuje 4 typy pamätí:

- Privátna pamäť je používaná iba jedinou výpočtovou jednotkou.
- Lokálna Pamäť môže byť použitá všetkými výpočtovými jednotkami v rámci jednej výpočtovej skupiny.
- Konštantná pamäť sa používa na uloženie konštantných dát pre „read-only“ prístup, dostupná všetkými výpočtovými jednotkami na zariadení počas vykonávania kernela. Hostovský procesor je schopný alokovať a inicializovať pamäťové objekty, ktoré sú uložené v tomto pamäťovom mieste.
- Globálna pamäť je dostupná pre všetky výpočtové jednotky na zariadení [1].



Obr. 6 Pamäťový model (OpenCL)

4 DirectCompute

DirectCompute je výpočtové API pre GPU od firmy Microsoft. DirectCompute je súčasťou DirectX10 a DirectX11 a podporuje všetky známe GPU podporujúce OpenCL, či CUDA.

Použitie DirectCompute:

- inicializácia DirectCompute,
- vytvorenie GPU kódu v HLSL jazyku,
- kompilácia použitím DirectX kompilera,
- nahranie kódu na GPU,
- nastavenie GPU buffera pre vloženie dát,
- vykonanie kódu na GPU,
- kopírovanie dát späť na pamäť CPU.

Viac informácií k tomuto rozhraniu v [10] a [5].

5 Optimalizácia výpočtov

5.1 Šírka pásma

Šírka pásma je jedným z kľúčových faktorov pre výkon. Takmer všetky zmeny v kóde by sa mali vykonávať s prihliadnutím na kontext, ako ovplyvnia šírku pásma. Šírka pásma môže byť veľmi ovplyvnená voľbou pamäte do ktorej sa ukladajú dáta, akým spôsobom sú dáta uložené, akým spôsobom sa k ním pristupuje a mnoho ďalších faktorov.

Pri meraní presnej výkonnosti je vhodné vypočítať teoretickú a efektívnu šírku pásma [22].

5.2 Kalkulácia teoretickej šírky pásma

Teoretická šírka pásma môže byť vypočítaná použitím hardvérovej špecifikácie dostupnej v produktovej literatúre. Napríklad karta použitá pri testovaní nVIDIA TESLA 1060 používa DDR pamäte s časovaním 800MHz a 512 bitov široké pamäťové rozhranie.

Použitím týchto informácií dokážeme vypočítať teoretickú šírku pásma pre nVIDIA TESLA c1060:

$$\left(900 \cdot 10^6 \times (512/\ 8) \times 2\right) \cdot 10^9 = 102{,}4\,GB/\text{sec}$$

V tejto kalkulácií je časovanie prevedené na Hz, vynásobené šírkou pamäťového rozhrania (vydelené 8 pre prevod na bajty) a vynásobené dvoma kvôli typu pamäti DDR (double data rate). Na záver bola výsledná hodnota prenásobená hodnotou 10^9 pre prevod výsledku na GB/sec [22].

5.3 Kalkulácia efektívnej šírky pásma

Efektívna šírka pásma sa vypočítava časovo špecifickými programovými aktivitami a znalosťami, akým spôsobom program pristupuje k dátam. Na tento účel slúži tento vzťah (výsledná šírka pásma je v GB/sec):

$$Efektívna\_šírka\_pásma = \frac{(B_r + B_w).10^9}{čas} \quad (1)$$

B_r - počet bajtov čítaných kernelom

B_w - počet bajtov zapísaných kernelom

čas - počet sekúnd

Napríklad na kalkuláciu efektívnej šírky pásma pre kopírovanie matice s rozmermi 2048 x 2048 sa použije nasledovný vzťah:

$$Efektívna\_šírka\_pásma = \frac{\left(2048^2 \times 4 \times 2\right).10^9}{čas} \quad (2)$$

Počet elementov matice je vynásobený veľkosťou každého elementu (4 byty pre float), vynásobené dvomi, pretože sa vykonáva čítanie aj zápis. Následne bola hodnota prenásobená hodnotou 10^9 pre prevod na výsledku na GB. Nakoniec sa výsledok predelí časom v sekundách, aby sme previedli výsledok na GB/sek [22].

5.4 Optimalizácie pamätí

Optimalizácie pamätí sú najdôležitejšou oblasťou pre zvýšenie výkonnosti aplikácií. Cieľom je maximalizovať použitie hardvéru maximalizáciou šírky pásma. Najefektívnejšie využitie šírky pásma je možné dosiahnuť použitím rýchlych pamätí vždy keď je to možné a naopak, používať tak málo pomalej pamäti, ako je to možné.

5.5 Kopírovanie údajov medzi host a device

Šírka pásma medzi device pamäťou a GPU je omnoho vyššia (102 GB/sek u nVIDIA TESLA c1060), než medzi host pamäťou a device pamäťou (8 GB/sek u PCI Express x16 Gen2). To znamená že je treba minimalizovať pohyb údajov medzi host a device, dokonca ak si to vyžaduje beh kernela na GPU, ktorý nevykazuje žiadne zrýchlenie v porovnaní s behom na CPU. To zahŕňa aj spojenie viacerých malých transferov medzi host a device pamäťami do jedného veľkého transferu.

Zložitejšie dátové štruktúry by sa mali vytvárať na device pamäti, práca s nimi prebiehať na device a mali by byť zrušené bez toho, aby boli mapované, alebo kopírované na host pamäti [22].

5.6 Zdieľaná pamäť a pamäťové bunky

Keďže je zdieľaná pamäť (`__local memory` pre OpenCL) uložená na čipe, je omnoho rýchlejšia než lokálna, alebo globálna pamäť. V skutočnosti je latencia zdieľanej pamäte 100x menšia, než latencia globálnej pamäte, teda v prípade že nenastali žiadne bunkové konflikty medzi vláknami.

Pre dosiahnutie vysokej šírky pásma pamätí pre zdieľaný prístup, zdieľaná pamäť je rozdelená na rovnako veľké pamäťové moduly, zvané bunky, ku ktorým sa dá pristupovať simultánne. Z toho dôvodu, každé ukladanie, alebo čítanie z *n* adries, ktoré ležia v *n* oddelených pamäťových bunkách, môže byť uskutočňované súčasne, s využitím šírky pásma, ktoré je *n* krát väčšia, ako šírka pásma jednej bunky.

Avšak, ak pochádza viacero adries z jednej pamäťovej bunky, tak je prístup serializovaný. Hardvér rozdelí požiadavky na pamäť, ktoré sú v konflikte, na toľko oddelených bezkonfliktných požiadaviek, koľko je potrebné. Tým sa znižuje efektívna šírka pásma o činiteľ, ktorý je rovný počtu separovaných pamäťových požiadaviek. Jediná výnimka je, ak všetky vlákna v polovičnom warpe adresujú na rovnaké miesto v zdieľanej pamäti, čo rezultuje v broadcast.

Pre minimalizovanie bunkových konfliktov je dôležité pochopiť ako pamäť adresuje dáta do pamäťových buniek. Bunky v zdieľanej pamäti sú usporiadané tak, že po sebe idúce 32 bitové slová sú priradené k po sebe idúcim bunkám a každá bunka má šírku pásma 32 bitov za takt. Šírka pásma zdieľanej pamäte je 32 bitov za takt.

U zariadení s výpočtovou schopnosťou 1.x má warp veľkosť 32 vlákien a počet buniek je 16. Požiadavka zdieľanej pamäte na warp je rozdelená na dve polovice, kde každá polovica spracováva jednu požiadavku. Na to, aby nedošlo k žiadnemu bunkovému konfliktu je potrebné, aby iba jedno pamäťové miesto v bunke bolo dostupné vláknami z polovice warpu [22].

6 Porovnanie CUDA a OpenCL

6.1 Programy na meranie výkonnosti

Program na meranie výkonnosti, všeobecné nazývaný Benchmark je štandardizovaný program, alebo súbor štandardizovaných programov, ktoré sa na zariadení spúšťajú za účelom testovania výkonnosti systému. Testom podlieha:

- hardvér (výkon CPU/GPU pri spracovávaní grafických operácií, rôznych typov matematických výpočtov a pod.),
- softvér (výkon databázových systémov, kompilerov, internetových prehliadačov a pod.).

Existuje však len veľmi málo benchmarkov pre testovanie výkonnosti paralelných výpočtov pre CUDA, alebo OpenCL. A aj to málo čo existuje, testuje iba výkonnosť akcelerátorov (hardvéru), neporovnáva tieto dve API voči sebe navzájom. Preto vznikla myšlienka urobiť porovnanie CUDA vs OpenCL. Výsledky testovania je možné vidieť v kapitole 6.6.

6.2 Analýza

Pre porovnanie CUDA a OpenCL bolo zvolené sčítavanie, násobenie a transponovanie štvorcových matíc, s rôznymi rozmermi a číselnými typmi. Praktické testy boli vykonané v rámci [6].

Za rozhodujúci faktor bol zvolený čas výpočtu. Merania prebehli na dvoch počítačových zostavách:

Zostava	Konfigurácia
1.	Intel(R) Core i3 – 2.93 GHz, 4 GB DDR, gr. nVIDIA GT240 a nVIDIA TESLA c1060
2.	Intel(R) Pentium D – 2.66 GHz, 1 GB DDR, gr. nVIDIA GeForce 8800GT

Ako vývojové a zároveň testovacie prostredie bolo zvolené Microsoft Visual Studio 2008. Ako jazykové prostredie používa CUDA modifikovaný jazyk C, takzvané CUDA C, ktoré využíva vstavaný C/C++ prekladač, v kombinácií so špecializovaným nvcc prekladačom. OpenCL pre host kód používa jazyk C++ a pre device kód modifikovaný jazyk C, tento krát však OpenCL. Ako prekladač používa device kód štandardný C/C++ prekladač, na preklad device kódu sa využíva CLCC prekladač. Testovanie CPU prebehlo v jazyku C++.

Keďže OpenCL umožňuje, aby bol výpočet uskutočnený na akcelerátoroch od rôznych výrobcov, tak sa predpokladá, že bude menej efektívny ako CUDA, ktorá je viazaná iba na akcelerátory od spoločnosti nVIDIA a je vyvíjaná spolu s hardvérom.

Obr. 7 Osadené nVIDIA TESLA c1060 a nVIDIA GT240 (KPI FEI TU Košice)

6.3 Násobenie matíc

Násobenie matíc je možné len vtedy, ak je počet stĺpcov ľavej matice rovnaký, ako počet riadkov pravej matice. Ak matica ***A*** je rozmeru *(m , n)* a matica ***B*** je rozmeru *(n , r)*, tak súčin týchto matíc ***C =A.B*** má rozmery *(m , n)* (*m* - počet riadkov v prvej matici, krát *n* - počet stĺpcov v druhej matici). Výsledná hodnota na pozícií [i,j] je:

$$\mathbf{C}_{[i,j]} = \mathbf{A}_{[i,1]}.\mathbf{B}_{[1,j]} + \mathbf{A}_{[i,2]}.\mathbf{B}_{[2,j]} + \ldots + \mathbf{A}_{[i,n]}.\mathbf{B}_{[m,j]} \quad (3)$$

pre každé *i* a *j*.

Pre testovanie boli použité matice typu n x n, kde n={16, 32, 64, 128, 256, 512, 1024, 2048, 4096}

6.3.1 Kernel

Kernel kódy pre CUDA (Obr. 8) a OpenCL (Obr. 9) sú veľmi podobné, líšia sa len v detailoch, ako je označenie kernela, premenných, (Tab. 2) či získavania ID jednotlivých vlákien (Tab. 3).

CUDA	OpenCL
__global__ funkcia	__kernel funkcia
__device__ funkcia	funkcia (bez označenia)
__constant__ deklarácia premennej	__constatnt deklarácia premennej
__device__ deklarácia premennej	__global deklarácia premennej
__shared__ deklarácia premennej	__local deklarácia premennej

Tab. 2 Rôzne označenia funkcií a premenných v rámci kernela

CUDA	OpenCL
threadIdx.x	get_local_id(0)
blockIdx.x * blockDim.x + threadIdx.x	get_global_id(0)
gridDim.x	get_num_groups(0)
blockIdx.x	get_group_id(0)
blockDim.x	get_local_size(0)
gridDim.x * blockDim.x	get_global_size(0)

Tab. 3 Rozdiely pri získavaní ID vlákien

```
//CUDA Kernel
__global__ void
matrixMul ( float* C, float* A, float* B, int wA)
{
//ziskavanie ID jednotlivych vlakien
int tx = threadIdx . x + blockIdx . x * blockDim . x;
int ty = threadIdx . y + blockIdx . y * blockDim . y;
//pomocna premenna, ktora uklada vysledky ziskane z vlakien
float sum = 0;
/nasobenie matic C = A * B
for ( int i = 0; i < wA; ++i)
{
       sum += A[ty * wA + i ] * B[i * wA + tx ];
}
//priradenie vysledku matici C
C[ty * wA + tx] = sum;
}
```

Obr. 8 CUDA kernel násobenia matíc

```
//OpenCL Kernel
__kernel void
matrixmul(__global float* C,
                __global float* A,
                __global float* B,
                int wA)
{
       //ziskavanie ID jednotlivych vlakien
       int tx = get_global_id (0);
       int ty = get_global_id (1);
//pomocna premena, ktora uklada vysledky z vlakien
float sum = 0;
//nasobenie matic C = A * B
for (int k = 0; k < wA; ++k)
{
      sum += A[ty * wA + k ] * B[k * wA + tx];
}
 //priradenie vysledku matici C
C[ ty * wA + tx] = sum ;
}
```

Obr. 9 OpenCL kernel násobenia matíc

Premenná wA označuje veľkosť strany matice A (width A) a keďže sú všetky matice štvorcové, s rovnakými rozmermi, tak wA vlastne popisuje rozmery všetkých matíc.

6.3.2 Optimalizácia výpočtov

Optimalizácia výpočtov spočíva v prenesení výpočtov z globálnej, do zdieľanej pamäte, tak ako je popísané v kapitole (5.4). Kernel je značne modifikovaný, no opäť sa kernele pre OpenCL a CUDA veľmi podobajú, z toho je uvedený iba kernel pre CUDA (Obr. 10).

```
//velkost bloku vlakien
#define BLOCK_SIZE 16

//CUDA Kernel
__global__ void
matrixMul ( float* C, float* A, float* B, int wA)
{
// ziskavanie ID blokov
int bx = blockIdx . x;
int by = blockIdx . y;
// ziskavanie lokalnych ID vlakien
int tx = threadIdx . x;
int ty = threadIdx . y;

// pomocna premenna, ktora pomaha ukladat vysledky ziskane z vlakien
float sum = 0;
// index prvej sub-matice matice A pocitanej blokom
int aBegin = wA * BLOCK_SIZE * by;
// index poslednej sub-matice matice A pocitanej blokom
int aEnd = aBegin + wA – 1;
// iteracny krok v matici A
int aStep = BLOCK_SIZE;
// index prvej sub-matice matice B pocitanej blokom
int aBegin = BLOCK_SIZE * bx;
// iteracny krok v matici B
int aStep = BLOCK_SIZE * wA;

// cyklus prechadzajuci vsetky sub-matice matic A a B
for ( int a = aBegin; b = bBegin;
            a <= aEnd
            a += aStep, b += bStep;)
{
//premenne operujuce v zdielanej pameti
_shared__ float As[BLOCK_SIZE] [BLOCK_SIZE];
_shared__ float Bs[BLOCK_SIZE] [BLOCK_SIZE];

//priradenie globalnych hodnot lokalnym premennym
As[ty][tx] = A[ a + wA * ty + tx];
Bs[ty][tx] = B[ b + wA * ty + tx];

//synchronizacia vlakien
__syncthreads ();

// nasobenie matic C = A * B
for ( int k = 0; k < BLOCK_SIZE; ++k)
      sum += As[ty][k] * Bs [k][tx];

//synchronizacia vlakien
__syncthreads ();

}
   //priradenie vysledku matici C
int c = wA * BLOCK _SIZE * by + BLOCK_SIZE * bx;
C[c + wA * ty + tx ] = sum;
}
```

Obr. 10 Optimalizovaný CUDA kernel násobenia matíc

Násobenie Matíc C = A * B (FLOAT)								
	16 × 16	32 × 32	64 × 64	128 × 128	256 × 256	512 × 512	1024 × 1024	2048 × 2048
CPU	0,0499	0,3839	3,0537	24,679	264,780	2587,89	25939,86	211606,23
CUDA	0,0770	0,0948	0,2968	2,6115	29,2306	170,6084	1026,3284	7574,0697
OpenCL	0,8844	0,9196	1,1575	3,6088	31,0733	162,2334	896,2493	6474,3230
OPT CUDA	0,0607	0,0668	0,0967	0,2394	0,9592	4,7490	30,0040	215,1323
OPT OpenCL	0,8172	0,8934	0,9502	1,2194	2,6781	9,6443	45,8106	292,6157
Zrýchlenie (neoptimalizovaná vs optimalizovaná verzia kernela)								
CUDA	1,3	1,4	3,1	10,9	30,5	35,9	34,2	35,2
OpenCL	1,1	1,0	1,2	3,0	11,6	16,8	19,6	22,1

Tab. 4 Zrýchlenie pri optimalizovaní násobenia matíc (GeForce 8800GT)

Násobenie Matíc C = A * B (DOUBLE)									
	16 × 16	32 × 32	64 × 64	128 × 128	256 × 256	512 × 512	1024 × 1024	2048 × 2048	4096 × 4096
CPU	0,067	0,338	2,673	21,138	175,54	1508,9	16977,6	146210,6	648301,8
CUDA	0,081	0,100	0,155	0,6156	4,8505	26,8452	158,293	1092,931	8789,222
OpenCL	0,737	0,774	1,070	2,1052	8,2336	34,4164	183,133	1204,431	8732,064
OPT CUDA	0,078	0,088	0,142	0,3874	1,5539	7,4804	45,3956	317,7468	2374,831
OPT OpenCL	0,800	0,936	1,034	1,9754	4,6786	17,9850	76,3719	441,0784	2904,615
Zrýchlenie (neoptimalizovaná vs optimalizovaná verzia kernela)									
CUDA	1,0	1,1	1,1	1,6	3,1	3,6	3,5	3,4	3,7
OpenCL	0,9	0,8	1,0	1,1	1,8	1,9	2,4	2,7	3,0

Tab. 5 Zrýchlenie pri optimalizovaní násobenia matíc (TESLA c1060)

Táto optimalizácia priniesla viditeľné zlepšenia výkonu už pri menších rozmeroch matíc, no skutočný význam sa ukázal až pri násobení matíc s väčšími rozmermi. Najviac sa táto optimalizácia prejavila u GeForce 8800GT, kde u CUDA, pri maticiach s rozmermi 2048 x 2048 prvkov, naplnených dátovým typom FLOAT, dosahuje až 35 násobné zrýchlenie oproti pôvodnému kernela a pri OpenCL je to 22 násobné zrýchlenie (Tab. 4). U všetkých kariet je pri dátovom type INTEGER, zaznamenaný pokles výkonnosti tejto optimalizácie a pri TESLA c1060 a dátovom type DOUBLE sa dostáva na hranicu 3 násobného zrýchlenia (Tab. 5).

6.3.3 Výsledky

Pri všetkých testovaných kartách a dátových typoch, si CUDA omnoho lepšie poradila s maticami menších rozmerov (Tab. 6). Ďalej sa ich výkonnosť vyrovnáva a pri GeForce 8800GT, OpenCL dokonca mierne prekonáva CUDA pri neoptimalizovaných výpočtoch s maticami nad 1024 x 1024 prvkov (Tab. 7) .

Násobenie Matíc C = A * B (FLOAT)								
	16 × 16	32 × 32	64 × 64	128 × 128	256 × 256	512 × 512	1024 × 1024	2048 × 2048
CPU	0,0803	0,3767	2,6679	21,386	166,83	1487,7	13896,1	118729,2
CUDA	0,1869	0,1984	0,2350	0,6072	3,3091	19,766	144,76	1166,154
OpenCL	4,5670	4,5546	4,5738	4,9284	8,2488	26,955	164,83	1225,591
OPT CUDA	0,2627	0,2249	0,2757	0,3608	0,8162	4,7109	30,194	226,5140
OPT OpenCL	4,5984	4,4079	4,4962	4,9961	6,1184	13,145	50,821	332,7690
CUDA vs OpenCL								
Normal	24,4	23,0	19,5	8,1	2,5	1,4	1,1	1,1
Optimalized	17,5	19,6	16,3	13,8	7,5	2,8	1,7	1,5

Tab. 6 Porovnanie OpenCL vs CUDA pri násobení matíc (GeForce 240GT)

Násobenie Matíc C = A * B (FLOAT)								
	16 × 16	32 × 32	64 × 64	128 × 128	256 × 256	512 × 512	1024 × 1024	2048 × 2048
CPU	0,0499	0,3839	3,0537	24,679	264,78	2587,89	25939,86	211606,23
CUDA	0,0770	0,0948	0,2968	2,6115	29,236	170,608	1026,328	7574,0697
OpenCL	0,8844	0,9196	1,1575	3,608	31,073	162,233	896,249	6474,3230
OPT CUDA	0,0607	0,0668	0,0967	0,2394	0,9592	4,7490	30,0040	215,13235
OPT OpenCL	0,8172	0,8934	0,950	1,2194	2,6781	9,6443	45,8106	292,61570
CUDA vs OpenCL								
Normal	11,5	9,7	3,9	1,4	1,1	1,0	0,9	0,9
Optimalized	13,4	13,4	9,8	5,1	2,8	2,0	1,5	1,4

Tab. 7 Porovnanie OpenCL vs CUDA pri násobení matíc (GeForce 8800GT)

6.4 *Sčítavanie matíc*

Sčítavanie dvoch matíc môže prebiehať len vtedy, ak tieto dve matice majú rovnaký rozmer. Sčítavajú sa čísla na rovnakých pozíciách.

$$\mathbf{C}_{[i,j]} = \mathbf{A}_{[i,j]} + \mathbf{B}_{[i,j]} \tag{4}$$

pre každé *i* a *j*.

Pre testovanie boli použité matice typu *n x n*, kde n={16, 32, 64, 128, 256, 512, 1024, 2048, 4096}

6.4.1 Kernel

Pri sčítavaní dvoch matíc sa používa pomerne jednoduchý algoritmus, ktorý je takmer identický pre OpenCL i CUDA.

```
//CUDA Kernel
__global__ void
matrixAdd ( float* C, float* A, float* B, int wA)
{
// Ziskavanie ID jednotlivych vlakien
int tx = threadIdx . x + blockIdx . x * blockDim . x;
int ty = threadIdx . y + blockIdx . y * blockDim . y;
// Scitavanie matic C = A + B
C [ ty * wA + tx] = A[ty * wA + tx] + B[ ty * wA + tx];
   }
```

Obr. 11 CUDA kernel sčítavania matíc

```
//OpenCL Kernel
__kernel void
matrixAdd (__global float* C,
__global float* A,
__global float* B,
int wA)
{
// Ziskavanie ID jednotlivych vlakien
int tx = get_global_id(0);
int ty = get_global_id(1);

// Scitavanie matic C = A + B
C [ ty * wA + tx] = A[ty * wA + tx] + B[ ty * wA + tx];
}
```

Obr. 12 OpenCL kernel sčítavania matíc

6.4.2 Optimalizácia výpočtov

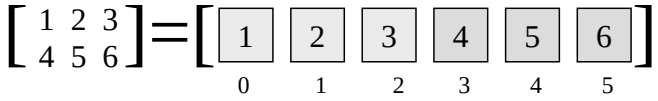
Optimalizácia výpočtov, podobne pri násobení matíc (6.3.2), spočíva v prenesení výpočtov z globálnej, do zdieľanej pamäte (Obr. 13).

```
// velkost bloku vlakien
#define TILE_SIZE 32
// CUDA Kernel
__global__ void
matrixAdd ( float* C, float* A, float* B, int wA)
{
//premenne operujuce v zdielanej pameti
      __shared__ float locA[TILE_SIZE] [TILE_SIZE];
      __shared__ float locB[TILE_SIZE] [TILE_SIZE];
// pomocna premenna pre scitanie
float sum = 0;
// ziskavanie globalnych ID vlakien
int gx = blockIdx . x + blockDim . x * threadIdx . x;
int gy = blockIdx . y + blockDim . y * threadIdx . y;
// ziskavanie lokalnych ID vlakien
int lx = threadIdx . x;
int ly = threadIdx . y;
// priradenie globalnych hodnot lokalnym premennym
locA[ly][lx] = A[ gy * wA + gx];
locB[ly][lx] = B[ gy * wA + gx];

// synchronizacia vlakien
__syncthreads ();
// scitavanie matic C = A + B
sum = locA[ly][lx] + locB[ly][lx];
// synchronizacia vlakien
__syncthreads ();

// priradenie vysledku matici C
C[gy*wA+gx] = sum;
}
```

Obr. 13 Optimalizovaný CUDA kernel sčítavania matíc

Táto optimalizácia však prináša iba mierne zrýchlenie výpočtov, omnoho kľúčovejším faktorom efektívneho sčítavania matíc sa ukázal spôsob, akým vyberáme prvky matice z poľa.

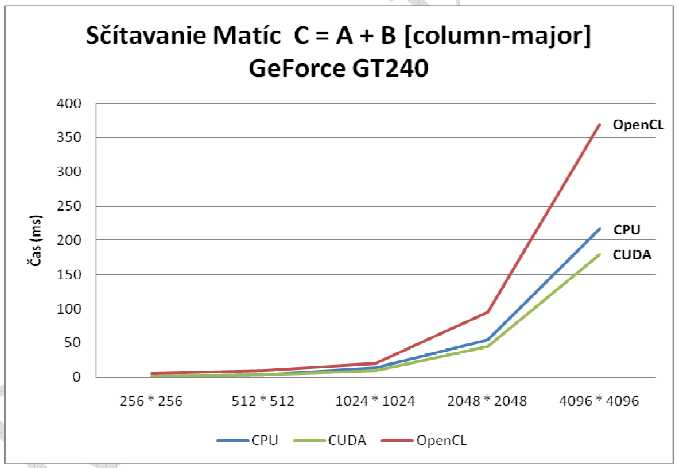
CUDA i OpenCL totiž viacrozmerné matice reprezentujú ako jednorozmerné pole. Rozhodujúcim sa teda ukazuje spôsob transformácie viacrozmernej matice do jednorozmerného poľa. Tu CUDA aj OpenCL používajú takzvanú *row-major* metódu, takže matice ukladajú do poľa po riadkoch. Príklad môžeme vidieť na obrázku (Obr. 14).



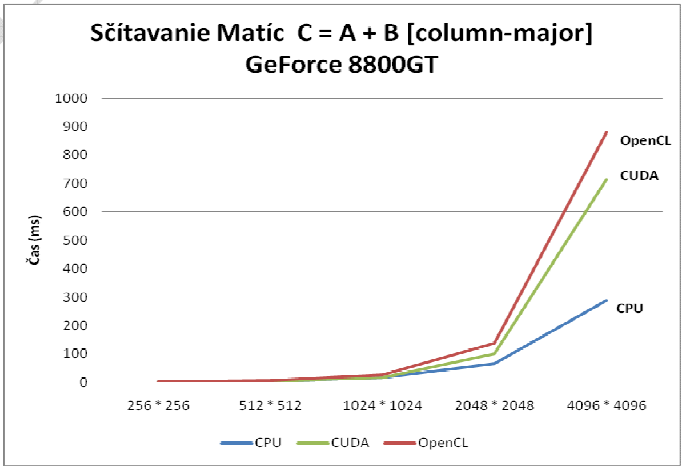
Obr. 14 Schéma rozkladu matice (row-major)

Ak teda vyberáme maticu po riadkoch, vyberáme prvky po rade, tak ako sú uložené, teda: [0, 1, 2, 3, 4, 5]. Ak však vyberáme prvé stĺpce matice (*column-major*), algoritmus musí skákať: [0, 3, 1, 4, 2, 5]. To má za následok, že efektivita rapídne klesá, keďže všetky moderné procesory sú optimalizované pre kontinuálne načítavanie dát, čo sa v tomto prípade mení.

Pri sčítavaní matíc, má takáto zlá práca s maticami za následok niekedy až viac ako trojnásobne spomalenie výpočtu. Pri výpočtoch s dátovými typmi INTEGER a FLOAT sa na všetkých testovaných kartách ukázalo, že u OpenCL je tento výpočet dokonca pomalší, ako sekvenčný výpočet na CPU (Obr. 15). Na kartách GeForce 8800GT a TESLA c1060, je pomalšia aj CUDA (Obr. 16).



Obr. 15Sčítavanie matíc GeForce GT240 (column-major)



Obr. 16 Sčítavanie matíc GeForce 8800GT (column-major)

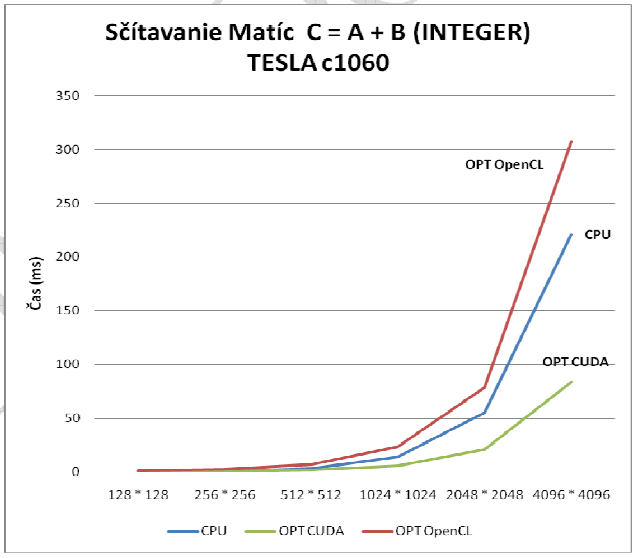
6.4.3 Výsledky

Pri sčítavaní matíc sú viditeľné rozdiely medzi CUDA a OpenCL, kde pri malých maticiach dosahuje CUDA 10 až 20 násobné zrýchlenie oproti OpenCL. Pri maticiach s rozmermi nad 1024 x 1024 prvkov je toto zrýchlenie vyššie ako 3 násobné (Tab. 8).

Sčítavanie Matíc C = A + B (INTEGER)									
	16 × 16	32 × 32	64 × 64	128 × 128	256 × 256	512 × 512	1024 × 1024	2048 × 2048	4096 × 4096
CPU	0,0070	0,0149	0,0524	0,2199	0,8570	3,4486	14,099	55,128	221,402
CUDA	0,0763	0,0739	0,0890	0,1785	0,6296	1,7950	5,5242	20,803	81,934
OpenCL	1,4712	0,7977	0,9079	1,2546	2,4957	7,1481	21,256	78,101	306,234
OPT CUDA	0,0695	0,0719	0,0890	0,1759	0,5172	1,8075	5,6285	21,193	83,445
OPT OpenCL	0,7903	0,8315	0,8609	1,3330	2,6790	7,1127	23,336	78,456	306,933
CUDA vs OpenCL									
Normal	19,3	10,8	10,2	7,0	4,0	4,0	3,8	3,8	3,7
Optimaliz.	11,4	11,6	9,7	7,6	5,2	3,9	4,1	3,7	3,7

Tab. 8 Porovnanie OpenCL vs CUDA pri sčítavaní matíc (TESLA c1060)

Pri sčítavaní matíc na akcelerátore TESLA c1060 sa ukázalo, že ani optimalizácia pomocou zdieľaných pamätí nepomohla OpenCL prekonať rýchlosť výpočtov na CPU (Tab. 8). Tieto rozdiely vo výkonnosti pri sčitovaní matíc sú zobrazené aj na obrázku (Obr. 17). Tu sa znova ukazuje, že CUDA dokáže efektívnejšie využívať potenciál paralelných výpočtov na akcelerátoroch nVIDIA.



Obr. 17 Porovnanie OpenCL vs CUDA pri sčítavaní matíc (TESLA c1060)

6.5 Transponovanie matíc

Pod transponovaním matice A=(a_{ij}) typu (m,n) rozumieme maticu $\mathbf{A}^T$=(b_{ji}) typu (n,m), kde $b_{ji} = a_{ij}$ pre každé i,j. Teda i-ty riadok matice **A** je i-tym stĺpcom matice $\mathbf{A}^T$ a j-ty stĺpec matice *A* je j-tym riadkom $\mathbf{A}^T$.

Pre testovanie boli použité matice typu n x n, kde n={16, 32, 64, 128, 256, 512, 1024, 2048, 4096}

6.5.1 Kernel

Pri algoritme transponovania matíc sa vymenia riadky matice za jej stĺpce (Obr. 18) (Obr. 19).

```
// CUDA Kernel
__global__ void
matrixAdd ( float* C, float* A, int wA)
{
// ziskavanie ID jednotlivych vlakien
int tx = threadIdx . x + blockIdx . x * blockDim . x;
int ty = threadIdx . y + blockIdx . y * blockDim . y;
// Transponovanie matic
C[ty * wA + tx] = A[tx * wA + ty]
}
```

Obr. 18 CUDA kernel transponovania matíc

```
// OpenCL Kernel
__kernel void
matrixAdd (__global float* C,
__global float* A,
int wA)
{
// ziskavanie ID jednotlivych vlakien
int tx = get_global_id (0);
int ty = get_global_id (1);

// Transponovanie matic
C[ ty * wA + tx] = A[tx * wA + tz];
}
```

Obr. 19 OpenCL kernel transponovania matíc

6.5.2 Optimalizácia výpočtov

Ako už bolo spomenuté v kapitole 6.4.2, práca s maticou, kedy vyberáme ako prvé stĺpce matice, značne spomaluje celkovú výkonnosť. Preto by bolo vhodné zmeniť v kernele (Obr. 18) algoritmus priradenia.

Pri optimalizácií transponovania matíc pomocou zdieľanej pamäte sa u OpenCL nepodarilo dosiahnuť ani minimálne zrýchlenie z dôvodu, že nebolo možné vykonať výmenu riadkov za stĺpce pomocou otočenia zdieľaných pamätí. U CUDA bolo možné nahrať do zdieľanej pamäte pôvodnú maticu „riadkami napred“ a pomocou výmeny súradníc bloku zdieľanej pamäte, nahrať rovnakým spôsobom transponovanú maticu späť do globálnej pamäte bez väčších časových strát (Obr. 20).

```
// velkost bloku vlakien
#define TILE_SIZE 16

// CUDA Kernel
__global__ void
matrixAdd ( float* C, float* A, int wA)
{
// premenna operujuca v zdielanej pameti
__shared__ float locA [TILE_SIZE][TILE_SIZE];


// ziskavanie globalnych ID vlakien
int gx = blockIdx . x * blockDim . x + threadIdx . x;
int gy = blockIdx . y * blockDim . y + threadIdx . y;

// ziskavanie lokalnych ID vlakien
int lx = thredIdx . x ;
int ly = thredIdx . y ;

// priradenie globalnych hodnot lokalnym premennym
locA[lx][ly] = A[gy*wA+gx]
```

```
// sybchronizacia vlakien
__syncthreads ();

// priradenie vysledku matici C
C[gy*wA+gx] = locA[ly][lx];
}
```

Obr. 20 Optimalizovaný CUDA kernel transponovania matíc

Takáto optimalizácia prináša 2 – 3 násobné zrýchlejnie výpočov, v porovnaní s neoptimalizovanou verziou kernela (Tab. 9).

Transponovanie matice $A = A^T$ (FLOAT)									
	16 × 16	32 × 32	64 × 64	128 × 128	256 × 256	512 × 512	1024 × 1024	2048 × 2048	4096 × 4096
CUDA	0,259	0,232	0,252	0,360	0,7126	2,3462	9,2908	48,711	297,01
OPT CUDA	0,212	0,231	0,251	0,348	0,5538	1,8072	6,2208	24,379	93,222
Zrýchlenie (neoptimalizovaná vs optimalizovaná verzia kernela)									
CUDA	1,2	1,0	1,0	1,0	1,3	1,3	1,5	2,0	3,2

Tab. 9 Zrýchlenie pri optimalizovaní transponovania matíc (GeForce 8800GT)

6.5.3 Výsledky

Keďže OpenCL neumožňuje efektívne využitie zdieľanej pamäte pre optimalizáciu transponovania matíc, do vzájomného porovnania CUDA a OpenCL táto nebola zahrnutá. No i tak sa opäť ukazuje, že CUDA dokáže lepšie pracovať so svojim hardvérom a prekonáva OpenCL na všetkých akcelerátoroch(Tab. 10).

Transponovanie matice $A = A^T$ (FLOAT)									
	16 × 16	32 × 32	64 × 64	128 × 128	256 × 256	512 × 512	1024 × 1024	2048 × 2048	4096 × 4096
CUDA	0,1786	0,1694	0,2187	0,2303	0,6136	1,3612	4,1924	18,845	74,235
OpenCL	3,5779	3,9114	4,2989	4,5149	5,2517	6,6482	11,637	42,824	162,98
CUDA vs OpenCL									
Normal	20,0	23,1	19,7	19,6	8,6	4,9	2,8	2,3	2,2
Optimaliz.	22,1	20,6	22,2	17,1	10,6	5,9	3,1	2,3	2,2

Tab. 10 Porovnanie CUDA pri transponovaní matíc (GeForce GT240)

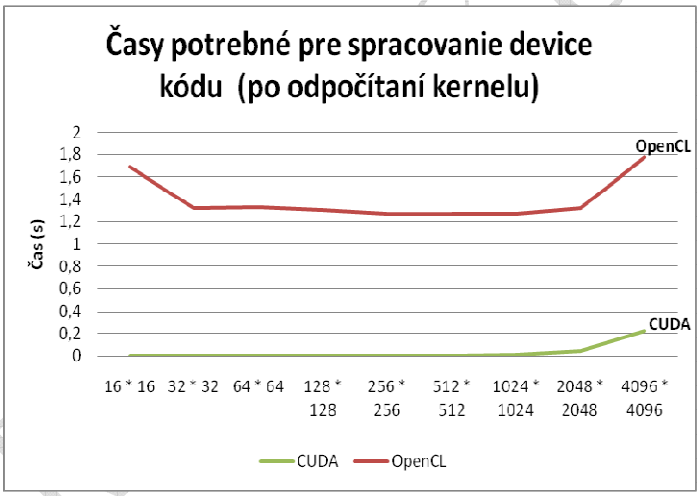
6.6 Zhrnutie

Pri násobení matíc s väčšími rozmermi, ako 2048 x 2048, sa narazilo na obmedzenie systému Windows, ktorý nepovolil vykonávanie kernela po dobu dlhšiu, ako po istý maximálny čas, ktorý sa približuje k 5 sekundám. Toto obmedzenie nastáva v prípade, keď systém MS Windows využíva jednu grafickú kartu (akcelerátor), ako pre grafické účely, tak aj pre všeobecné výpočty. Ak je teda využívaný akcelerátor pre výpočty dlhšie, ako je táto maximálna hranica, MS Windows zastaví všetky procesy využívajúce daný akcelerátor, v tomto prípade kernel.

Toto obmedzenie sa dá prekonať použitím viacerých akcelerátorov, kde jeden akcelerátor spracováva grafické operácie, nutné pre chod systému a druhý akcelerátor spracováva všeobecné výpočty. Pri testoch to bola kombinácia akcelerátorov GeForce GT240 (grafické operácie) a TESLA c1060 (všeobecné výpočty).

Zaujímavé je taktiež porovnanie celkových časov potrebných na vykonanie device kódu, po odrátaní času potrebného na výpočet (kernel). Tento čas vlastne zahŕňa všetky inicializácie nutné pre chod OpenCL, respektíve CUDA. Ako je vidieť na obrázku (Obr. 21), čas potrebný pre inicializáciu OpenCL sa pohybuje v priemere okolo 1,3 sekundy. Čas potrebný pre inicializovanie CUDA je zanedbateľný a výraznejšie rastie až pri veľkých rozmeroch, kedy musí inicializovať a spracovávať matice s veľkými rozmermi.

Z hľadiska porovnania akcelerátorov boli použité tieto: nVIDIA GeForce 8800GT, nVIDIA GeForce GT240 hlavne nVIDIA TESLA c1060.



Obr. 21 Časy potrebné pre spracovanie device kódu CUDA a OpenCL

nVIDIA GeForce 8800GT bol zo všetkých testovaných akcelerátorov najstarší. Jedná sa vlastne o prvý model, ktorý obsahoval programovateľné CUDA jadrá. Podľa očakávaní je najpomalší vo všetkých testovaných odvetviach. Ako jediný dokázal efektívnejšie pracovať s OpenCL, než s CUDA, pri neoptimalizovanom násobení matíc (Tab. 7). nVIDIA GeForce GT240 ako najnovší akcelerátor zo všetkých testovaných sa ukázal ako najvýkonnejší pri jednoduchých výpočtoch, akými sú sčítavanie, či transponovanie matíc. nVIDIA TESLA c1060 je jediným zástupcom rady špecializovaných akcelerátorov navrhnutých výhradne pre spracovávanie všeobecných výpočtov. Tento akcelerátor neobsahuje výstup pre monitor, jedná sa teda o čisto výpočtovo orientované zariadenie. Ako jediný akcelerátor je taktiež schopný uskutočňovať výpočty vo zvýšenej presnoti, takzvanej double precision. TESLA c1060 vyniká pri násobení matíc, kde prekonáva ostatné akcelerátory.

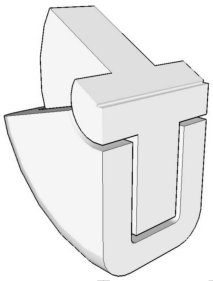
Nepreukázali veľké rozdiely pri spracovávaní dátových typov float a integer. Iba pri násobení matíc sa ukázalo, že akcelerátorom sa lepšie pracuje s číslami v pohyblivej rádovej čiarke. To malo za následok, že pri práci so zdieľanou pamäťou dosahovali výpočty s dátovým typom float, 2 krát taký výkon, ako pri páci s integermi.

Dátový typ double sa v GPGPU používa na výpočty vo zvýšenej presnosti. Aby bolo možné počítať v tejto zvýšenej presnosti, je nutné vykonať niekoľko úprav v programe:

- CUDA – do príkazového riadka pre nvcc je treba pridať `"--gpu-name sm_13"`
- OpenCL – do device kódu je treba pridať direktívu "`#pragma OPENCL EXTENSION cl_khr_fp64 : enable`" tak, aby bola umiestnená ešte pred deklaráciou akéhokoľvek dátového typu double v kernele.

Vo všeobecnosti sa ukázalo, že výpočty vo zvýšenej presnosti sú v priemere 2 až 3 krát pomalšie, ako výpočty v jednoduchej presnosti, teda s dátovým typom float.

Pri takmer všetkých druhoch výpočtov, so všetkými akcelerátormi sa ukázalo, že CUDA dokáže efektívnejšie využiť výpočtový potenciál akcelerátorov nVIDIA. Jedinou výnimkou je akcelerátor nVIDIA GeForce 8800GT, ktorý pri neoptimalizovanom násobení matíc dosahoval mierne zvýšenú efektivitu OpenCL (Tab. 7). Takže sa potvrdil predpoklad, že OpenCL bude vďaka svojej všestrannosti strácať a naopak, CUDA, ktorá je vyvíjaná spoločnosťou nVIDIA, spolu s hardvérom, je na tento hardvér lepšie prispôsobená a dokáže lepšie využiť jeho potenciál.



7 Literatúra

[1] Sobota, B., Perháč, J., Straka, M., Szabó, Cs.: Aplikácie paralelných, distribuovaných a sieťových počítačových systémov na riešenie výpočtových procesov v oblasti spracovania rozsiahlych grafických údajov; elfa Košice, 2009, ps. 180, ISBN 978-80-8086-103-2

[2] Sobota, B., Korečko, Š., Látka, O., Szabó, Cs., Hrozek, F.: Riešenie úloh spracovania rozsiahlych grafických údajov v prostredí paralelných počítačových systémov; UK TU Košice, 2012, ps. 378, ISBN 978-80-553-0864-7

[3] AMD. An Introduction to OpenCL [online]. Advanced Micro Devices, Inc. 2011, url: <http://www.amd.com/>.

[4] BrookGPU, 2008, url: http://graphics.stanford.edu/projects /brookgpu/index.html

[5] Class, Gus. DirectCompute Lecture Series 101: Introduction to DirectCompute [online]. Microsoft, 2011, url: <http://channel9.msdn.com/>.

[6] Filin, M.: Využitie akcelerátorov vo vedeckotechnických výpočtoch. Diplomová práca. Košice: FEI Technická univerzita v Košiciach, 2011

[7] Khronos Group. About the Khronos Group [online]. Khronos Group, 2011. url: <http://www.khronos.org/>.

[8] Khronos Group. OpenCL Reference Pages [online]. Khronos Group, 2011. url : <http://www.khronos.org/>.

[9] Khronos OpenGL, 2011, url: http://www.opengl.org

[10] Microsoft Corp.: Direct Compute, url: http://www.microsoft. .com/download/en/details.aspx?displaylang=en&id=16995, 2011

[11] Moore G: Cramming more components onto integrated circuits, Electronics, vol. 38, no. 8, Apr. 1965.

[12] nVIDIA Corporation. DirectCompute [online]. nVIDIA Corporation, 2011, url: <http://www.nVIDIA.com/>.

[13] nVIDIA Corporation. NV Occlusion Query Extension, jún 2008. BIBLIOGRAPHY 109 http://oss.sgi.com/projects/ogl-sample/registry/NV/

[14] nVIDIA Corporation. nVIDIA GeForce 8800 GT [online]. nVIDIA Corporation, 2011, url: <http://www.nVIDIA.com/>.

[15] nVIDIA Corporation. nVIDIA GeForce GT 240 [online]. nVIDIA Corporation, 2011, url: <http://www.nVIDIA.com/>

[16] nVIDIA Corporation: nVIDIA CUDA Architecture: Introduction & Overview. Santa Clara : nVIDIA Corporation, 2009.

[17] nVIDIA Corporation: nVIDIA CUDA C: Programmin Guide. Santa Clara : nVIDIA Corporation, 2010.

[18] nVIDIA Corporation: nVIDIA CUDA Compute Unified Device Architecture - Programming Guide, 2011, url: http://developer.nVIDIA.com/cuda

[19] nVIDIA Corporation: nVIDIA TESLA C1060 Computing Processor Board. Santa Clara : nVIDIA Corporation, 2010.

[20] nVIDIA Corporation: nVIDIA Tesla: High Performance Computing, 2011, url: www.nVIDIA.com/object/tesla_computing_solutions.html

[21] nVIDIA Corporation: nVIDIA's Next Generation CUDA Compute Architecture: Fermi. Santa Clara : nVIDIA Corporation, 2009.

[22] nVIDIA Corporation: OpenCL: Best pactices Guide. Santa Clara : nVIDIA Corporation, 2010.

[23] nVIDIA SLi, 2011, url: http://www.nVIDIA.com/page/sli.html

[24] Pharr M., Fernando R.: GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley Professional, 2005, ISBN 0321335597