VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

VYUŽITÍ GRAFICKÉHO ADAPTÉRU PRO OBECNÉ VÝPOČTY

GENERAL-PURPOSE COMPUTATION USING GRAPHICS CARD

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE MICHAL BOČEK
AUTHOR

VEDOUCÍ PRÁCE Ing. JIŘÍ JAROŠ
SUPERVISOR

BRNO 2010

Abstrakt

Tato bakalářská práce se zabývá popisem programovacích modelů OpenCL a CUDA pro paralelní programování grafických adaptérů a v případě OpenCL i dalších výpočetních platforem. Je zde popsána implementace aplikace počítající elektrický potenciál v krystalické mřížce. Použitý algoritmus byl naprogramován s využitím dvou technologií pro GPU - OpenCL a CUDA. Byla mezi nimi porovnána rychlost výpočtu a dále s rychlostí výpočtu na CPU.

Abstract

This thesis describes the programming models OpenCL and CUDA for Parallel Programming adapters and in case of OpenCL even for other computing platforms. There was implemented the application which calculates the electric potential in the crystalline lattice. The algorithm was programmed using two technologies for the GPU - OpenCL and CUDA. Their computational time were compared together with computational time of the CPU.

Klíčová slova

OpenCL, CUDA, GPGPU, výpočet elektrického potenciálu.

Keywords

OpenCL, CUDA, GPGPU, electric potential computation.

Citace

Boček Michal: Využití grafického adaptéru pro obecné výpočty, bakalářská práce, Brno, FIT VUT v Brně, 2010

Využití grafického adaptéru pro obecné výpočty

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jiřího Jaroše. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Michal Boček
19. 5. 2010

Obsah

1 Úvod

Tato bakalářská práce se zabývá popisem programovacích modelů OpenCL a CUDA pro paralelní programování grafických adaptérů. Je zde popsána implementace aplikace počítající elektrický potenciál v krystalické mřížce. Použitý algoritmus byl naprogramován s využitím dvou technologií pro GPU - OpenCL a CUDA. Byla mezi nimi porovnána rychlost výpočtu a dále s rychlostí výpočtu na CPU.

Text práce je rozdělen na následující části: kapitola 2 pojednává o historii GPGPU, výhodách a nevýhodách. Kapitola 3 vysvětluje použití knihoven CUDA a OpenCL. Kapitola pět popisuje implementaci programu, jehož výsledky jsou potom v kapitole pět porovnány.

2 GPGPU

GPGPU je souhrné označení pro využití grafických karet pro obecné výpočty. GPU umožňuje provádět vysoce paralelní výpočty na velkém počtu výpočetních jader, a to je činí více efektivní oproti CPU ve zpracování určitých algoritmů. Původně byly GPU používané pro zpracování 3D grafických dat, na což byly optimalizovány. S příchodem programovacích modelů (sady softwarových technologií - aplikace, programovací jazyky, překladače, knihovny), mezi něž patří NVIDIA CUDA nebo OpenCL, které zjednodušují vývoj paralelního programování, se stává GPU mnohem použitelnější pro účely nesovisející jen s grafickými výpočty.

2.1 Paralelní programování

Paralelní architektura procesorů a programovací modely nejsou nové, jeden z prvních paralelních počítaču byl Cray 1 z roku 1976. Nerozšířily se ale masově, zůstaly u superpočítačů pro výpočty složitých úkolů, protože vývoj paralelních programů byl a prozatím i je podstaně dražší než vývoj programů sekvenčních. Stále je potřeba mnohem více času a úsilí k vytvoření takového programu. Pro vytvoření paralelního programu musí totiž vývojář:

- pochopit konkurentnost procesů, jejich synchronizaci a komunikaci mezi nimi,
- použití datových struktur umožňující paralelní zpracování,
- pro efektivní aplikaci znát hardware, na kterém bude program vykonáván.

V poslední době se ale paralelní programování stává čím dál více rozšířené. Jedním důvodem je expandující výroba vícejádrových CPU, dalším důvodem je, že výrobci grafických čipů vylepšují programovacími modely, které programování GPGPU zjednodušují. Takovými jsou například NVIDIA CUDA, OpenCL nebo ATI Stream. Musíme vědět, komu to chceme říci

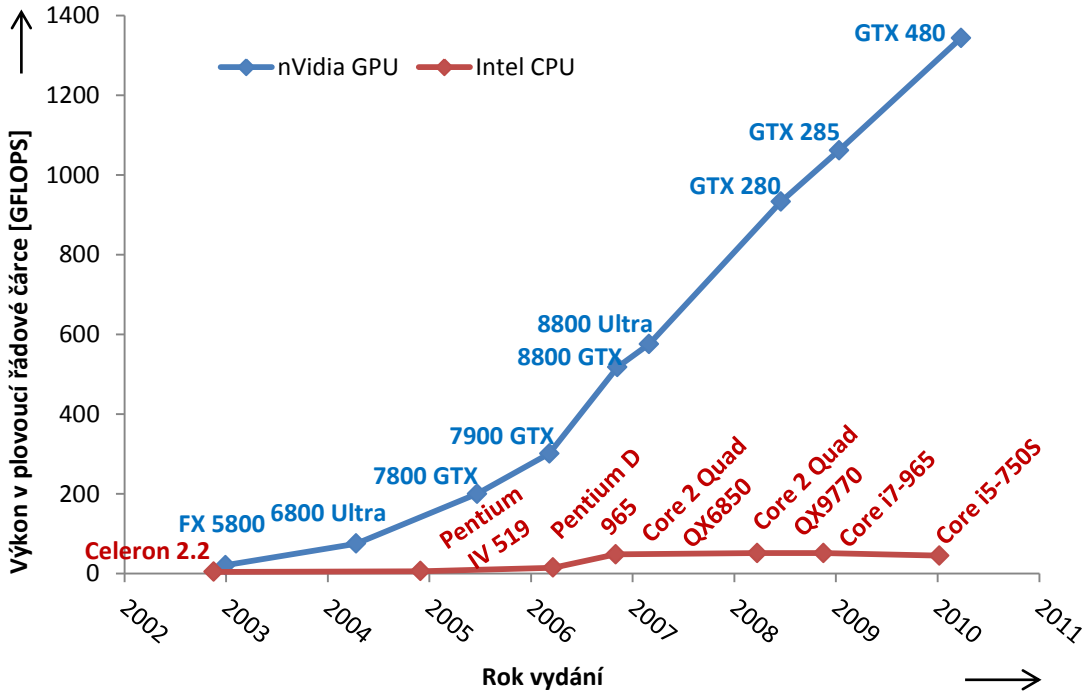
Dalším důležitým předpokladem dobrého psaní je *psát pro někoho*. Píšeme-li si poznámky sami pro sebe, píšeme je jinak než výzkumnou zprávu, článek, diplomovou práci, knihu nebo dopis. Podle předpokládaného čtenáře se rozhodneme pro způsob psaní, rozsah informace a míru detailů.

2.2 Důvody pro přechod z CPU na GPU

Hlavním důvodem je výkon GPU v určitých typech aplikací. GPU je optimalizováno převážně pro aritmetické operace v plovoucí desetinné čárce a pro práci s poli a vektory. Na rozdíl od CPU má méně vnitřní logiky, např. pro obsluhu přerušení, pro ochranu paměti, stejně tak pro různé dynamické předvídání instrukcí. Na GPU se uplatní nejvíce algoritmy provádějící výpočty v plovoucí řádové

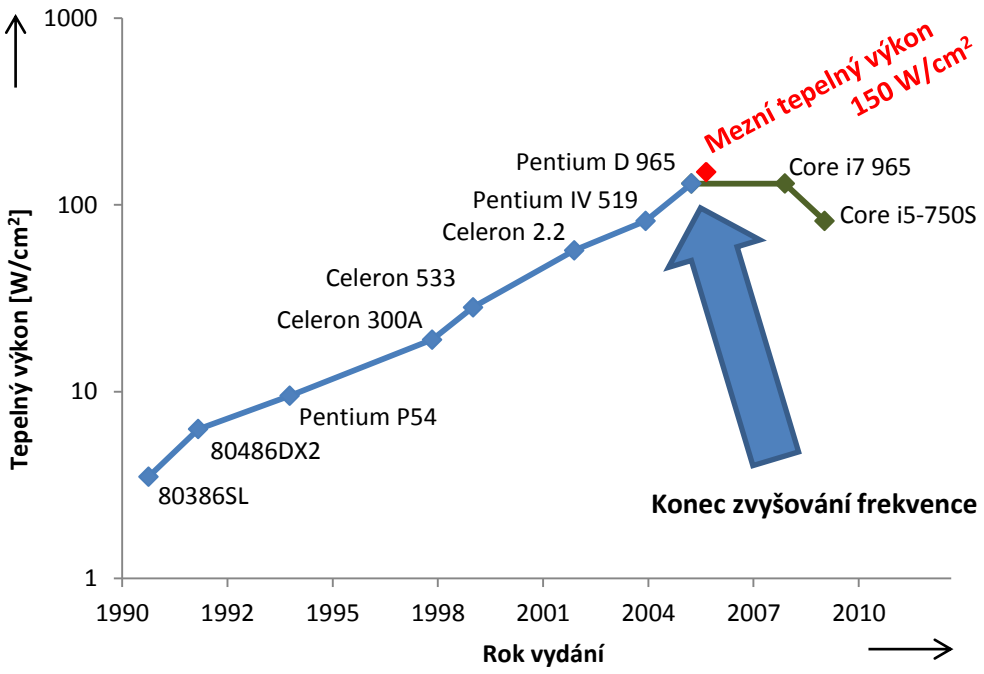
čárce a paralelně na vysokém počtu dat. Jde o tzv. high performance comuting (HPC). Na druhou stranu v těchto výpočtech je GPU mnohonásobně výkonnější než CPU. Hlavní rozdíl je v počtu výpočetních jednotek. Zatímco současná moderní CPU mají <10 jader, na GPU je jich <1000. Jednotlivá jádra na GPU jsou sice méně výkonná, ale využívají se současným výpočtem menších programů najednou. Dále je zde potřeba zmínit paměťovou propustnost. GPU pracuje se svojí pamětí DRAM na několikanásobně vyšší rychlosti, než CPU. Nejnovější NVIDIA GTX480 má maximální paměťovou propustnost 177,4 GB/s oproti 25,6 GB/s u procesoru Intel i7-965.

V roce 2010 bylo prodáno již přes 200 milionů grafických karet podporující CUDA a je zde tedy velký potenciál trhu pro vývoj rychlejších verzí stávajících sekvenčních programů a pro výrobce to znamená neustálé vylepšování nástrojů pro využití těchto karet.



Graf 1 Zvětšování rozdílu ve výkonu GPU a CPU

Dalším důvodem je výrazné zpomalení zvyšování výkonu CPU. Mooreův zákon, předpovídající zdvojnásobení počtu tranzistorů na čipu každé dva roky, stále platí. Každý tranzistor má ale jisté tepelné vyzařování a vysoký počet tranzistorů na plošnou jednotku a jejich tepelný výkon dosáhl již takové míry, že již není možné zvyšovat frekvenci. Zvyšování frekvence CPU bylo ukončeno v roce 2004, kdy byl pro vysoké provozní teploty ukončen vývoj procesoru Intel Tejas (prototypován na 7 GHz). Podle profesora Rabaey z Univerzity v Berkeley je mezní hranicí tepelného výkonu procesoru hodnota 150 W/cm^2, za kterou nastávají selhání harwaru [10].



Graf 2 Zvyšování tepelného výkonu CPU

Jelikož již nešlo zvyšovat frekvenci CPU, bylo potřeba změnit jeho architekturu. Změna architektury se projevila ve využití vysokého počtu tranzistorů na čipu k implementaci více výpočetních jader CPU a z toho vyplývající paralelní zpracování. Nejnovější procesor od Intelu Xeon X7560 má 8 jader. Tím se CPU blíží architektuře mnohojádrových GPU. Nejnovější GPU NVIDIA GTX480 má 480 jader. Budoucí vývoj aplikací jak pro CPU, tak pro GPU tedy směřuje k paralelnímu zpracování. Již existují prototypy procesorů spojující architekturu GPU s CPU (Intel Larabee a AMD Fusion), které jsou důkazem toho, jaké úspěchy již programování GPU dosáhlo.

Nevýhody:

- Omezující rychlost přenos dat mezi zařízením a hostem.
- GPU nejsou moc chytré, zpracovávají obtížně chyby.
- Aplikace je složitější debugovat.
- Data musí být specificky optimalizována pro efektivní běh aplikace.

3 Knihovny

Knihovny, kterým se budu věnovat, jsou CUDA, OpenCL a DirectCompute. DirectCompute je alternativou od Microsoftu, je to podmnožina DirectX API pro využití GPU pro obecné výpočty. Výhodou DirectCompute je nezávislost na výrobci GPU, nevýhodou je ovšem programovatelnost pouze pod OS Windows verze Vista SP2 a výšší a nedostatek dokumentace a příkladů.

CUDA a OpenCL jsou si velice podobné architektury, poskytující podobnou funkcionalitu. OpenCL víceméně vycházelo z CUDA při svém vývoji s tím, že cílem bylo eliminovat nedostatky CUDA, jako je spustitelnost CUDA aplikace pouze na zařízení od firmy NVIDIA. OpenCL je otevřený standard, podporující programování nejen GPU, ale i CPU, DSP a dalších typů procesorů.

3.1 CUDA

CUDA je název firmy NVIDIA pro paralelní výpočetní hardwarovou architekturu. Byla představena v listopadu 2006 při uvedení čipu G80. Zařízení podporující CUDA jsou pouze ta od firmy NVIDIA. Základem pro programování CUDA je CUDA Toolkit. Dále je potřeba mít nainstalovaný driver.

Charakteristiky CUDA:

- Jednotné řešení pro paralelní programování NVIDIA grafických procesorů.
- Podpora standardního C/C++ programovacího jazyka.
- Optimalizovaný přenos dat mezi CPU a GPU.
- Spolupráce s grafickým API - OpenGL a DirectX.
- Podpora 32- a 64-bitových operačních systémů Windows, Linux i MacOS X.
- Výpočty s jednoduchou i dvojitou přesností v plovoucí řádové čárce.
    - Výpočty s dvojitou přesnost jsou dostupné až od Compute Capability 1.3 (viz Příloha A) a jsou několikrát pomalejší než s jednoduchou.
- Nízkoúrovňové i vysokoúrovňové programování.

Podpora double-precision výpočtu se zdála být při uvedení (od GTX200) pouze jako populismus, vzhledem k tomu, že si mnoho vědců stěžovalo, že potřebují počítat přesnější výsledky a proto nechtějí přejít na GPGPU. Rychlost double-precision oproti single-precision byla ale 8x nižší. Uvedením GTX480 tato se tento rozdíl ovšem snížil na čtvrtinu, tedy že výpočty double-precision jsou jen 2x pomalejší. To již může mnoho lidí přesvědčit. Ovšem rozdíl zde pořád je, a proto jsou single-precision výpočty hlavní sílou GPU.

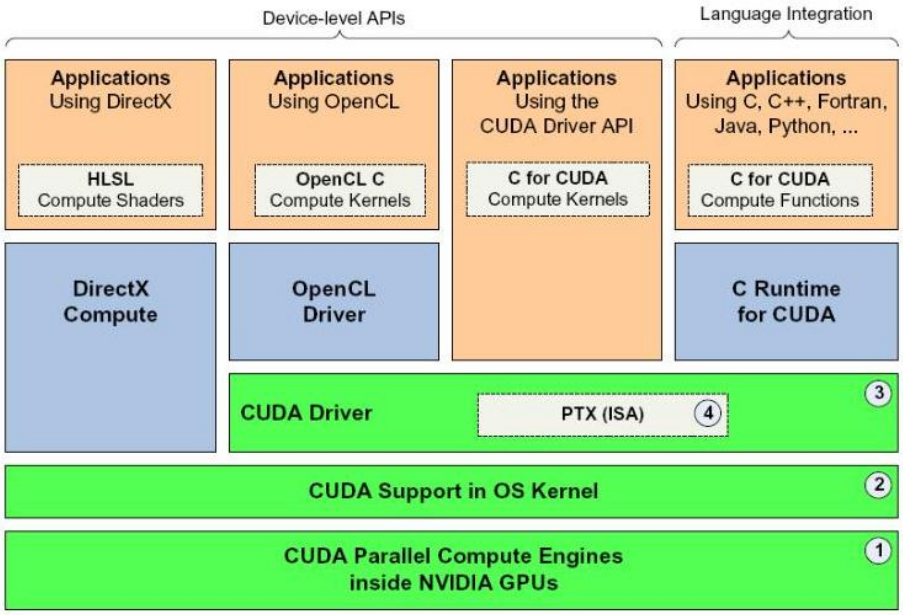
3.1.1 CUDA Toolkit

NVIDIA poskytuje kompletní balík nástrojů - CUDA Toolkit - pro programování CUDA, který obsahuje

- kompilátor (nvcc),
- profiler (cudaprof),
- debugger (cudagdb),
- dokumentaci,
- knihovny,
    - cuBLAS – práce s vektory,
    - cuFTT – implementace rychlé Fourierova Transformace,
    - NPP – práce s obrazem a videem
- ukázky kódu.

CUDA Toolkit umožňuje programovat kernel více možnostmi. Jednou je programování aplikace na nízké úrovni pomocí OpenCL, DirectCompute a CUDA Driver API. Další je využití standardního jazyka C s CUDA rozšířeními, které využívá rozhraní C (C Runtime for CUDA) vyvinuté pro ulehčení programování CUDA zařízení. Toto rozhraní poskytuje kompletní podporu jazyka C a umožňuje i propojení s vysokoúrovňovými jazyky, jako je Python, Java a C++.

Při programování v jazyce C s CUDA rozšířeními je výhodou možné využití existujících sekvenčních kódů pro CPU psaných v C a přepsat jen výpočetně náročné funkce pro vykonání na GPU.



Obrázek 1 Použitelné jazyky. Převzato z [8]

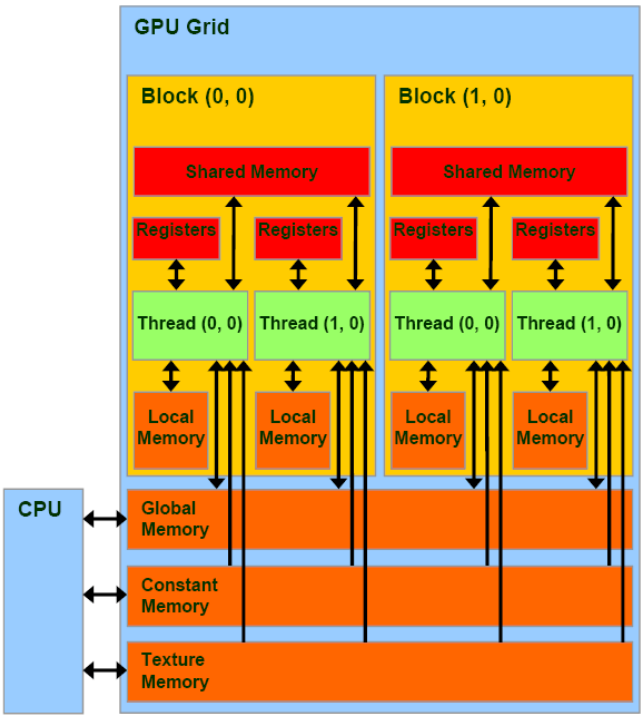
3.1.2 Architektura CUDA a její programování s jazykem CUDA C

Architektura zařízení

Každé zařízení (odpovídá GPU v terminologii CUDA) je rozděleno na několik streamovacích multiprocesorů (Streaming Multiprocessor – SM). Jejich počet je různý na různých generacích zařízení. Každý z multiprocesorů obsahuje několik streamovacích procesorů (Streming processor – SP). Nejnovější NVIDIA 480GTX obsahuje 480 SP (15 multiprocesorů, každý s 32 SP) Každé GPU obsahuje až 4 GB paměti DRAM, označovanou jako globální paměť.

Paměťový model

Procesor CPU (v terminologii CUDA host) a GPU (zařízení, device) má každý svojí paměť. Zařízení NVIDIA má několik typů pamětí, každá je vhodná pro jiný typ použití. Těmito typy jsou globální - global, lokální - local, sdílená - shared, texturová - texture and registry - registers, znázorněné na následujícím obrázku.



Obrázek 2 Paměťový model CUDA [7]

Typ paměti	Cachovaná	Přístup z kernelu	Rozsah	Životnost
Registr	není potřeba - na čipu	Čtení/zápis	1 vlákno	Vlákno
Lokální	Ano (compute capability > v2.0)	Čtení/zápis	1 vlákno	Vlákno
Sdílená	není potřeba - na čipu	Čtení/zápis	Všechna vlákna v bloku	Blok
Globální	Ano (compute capability > v2.0)	Čtení/zápis	Všechna vlákna a CPU	CPU alokace/uvolnění
Konstantní	Ano	Pouze čtení	Všechna vlákna a CPU	CPU alokace/uvolnění
Texturová	Ano	Pouze čtení	Všechna vlákna a CPU	CPU alokace/uvolnění

Tabulka 1 Typy pamětí

Globální, lokální a texturová paměť mají nejvyšší přístupové doby, rychlostí následující jsou konstantní a sdílená paměť a registry.

Globální paměť

V podstatě hlavní paměť, sestávající z většiny kapacity DRAM pro GPU. Přístup na ní je ovšem velice pomalý. Pro co nejvyšší rychlost je potřeba přistupovat k sousedním paměťovým místům

zároveň – tzv. memory coalescing. Je to tím, že moderní DRAM při přístupu na paměťové místo zároveň čte paralelně i několik sousedních paměťových míst, až 128 B. Proto pokud se vyžádají programem tyto sousední místa, jsou vrácena za zlomek času, protože jsou na DRAM přečtena najednou. Globální paměť je cachovaná pro zařízení s Compute Capability od verze 2.0 výše.

Sdílená paměť

Protože je umístěná na čipu, je mnohem rychlejší než lokální a globální paměť a to přibližně 100-150x. Je přístupná všem vláknům v rámci bloku a proto jim dovoluje mezi sebou přes tuto paměť spolupracovat.

Lokální paměť

Každé vlákno má svojí lokální paměť. Ta ovšem není přímo na čipu a proto je přístup na ní stejně pomalý jako do globální paměti. Jako globální paměť i lokální paměť není cachovaná na zařízeních s Compute Capability 1.x. Lokální paměť se použije automaticky kompilerem, když vlákno vyčerpá všechny registry pro proměnné.

Konstantní paměť

Konstantní paměť je cachovaná, tudíž při čtení jde o přístup do DRAM, když není hodnota v cache, jinak jde o přístup do velmi rychlé cache. Čtení z této cache je ideálně stejně rychlé jako čtení z registrů, a to pokud všechny vlákna ve warpu (32 vláken) čtou ze stejné adresy. Jinak se rychlost zmenšuje lineárně s počtem různých adres, z kterých vlákna čtou v jeden čas.

Texturová paměť

Texturová paměť je také cachovaná jako konstantní. Texturová cache je ovšem více optimalizovaná na přístup k 2D polím, tudíž vlákna z jednoho warpu, která čtou z adres co nejvíce blízko sebe, získají nejvyšší výkon. Je tedy nejvíce vhodná pro zpracvání obrazu.

Registry

Jde o nejrychlejší možnou paměť využitelnou vláknem. Kompilátor do ní ukládá proměnné.

Funkce pro práci s pamětí

CUDA obsahuje C funkce pro práci s pamětí GPU, které použitím odpovídají funkcím jazyka C:

- alokace - `cudaMalloc()`
    - `cudaMalloc((void**)&potentialMapOnDevice, memorySizeForPotentialMap);`
- nastavení - `cudaMemset()`
    - `cudaMemset(potentialMapOnDevice, 0, memorySizeForPotentialMap);`
- kopírování - `cudaMemcpy()`

- je zde potřeba specifikovat, zda se kopíruje z CPU na zařízení nebo obráceně
- `cudaMemcpy(potentialMapOnHost, potentialMapOnDevice, memorySizeForPotentialMap, cudaMemcpyDeviceToHost)`

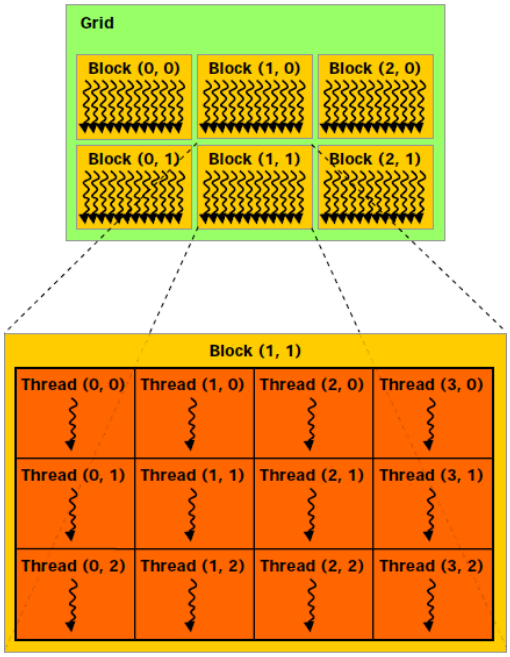
- uvolnění – `cudaFree()`
  - `cudaFree(potentialMapOnDevice);`

Vlákno – blok - grid

Vlákno (thread) je jeden proces, zpracovávaný na GPU. Jedno vlákno GPU je extrémně malé oproti vláknům CPU a mezi jednotlivými vlákny funguje velmi rychlé přepínání. Těchto vláken je potřeba pro dosažení vysoké efektivity mít desetitisíce.

Vlákna jsou seskupena do **bloků** (block, thread block). Blok má sdílenou pamět (shared memory), ke které mohou přistupovat všechna vlákna daného bloku. Vlákna různých bloků spolu ovšem nemohou nijak komunikovat. V rámci bloku je možné synchronizovat spouštění vláken (funkce `syncthreads()`), např. až všechny vlákna zpracují určitý mezivýsledek uprostřed vykonávání vlákna, bez kterého nelze počítat dále.

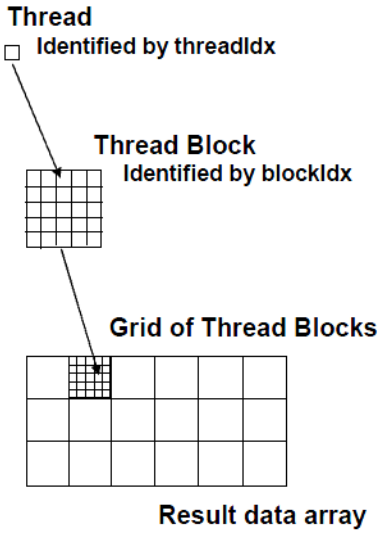
Gridem se nazývají všechna vlákna, která vzniknou při spuštění jednoho kernelu.



Obrázek 3 Hierarchie vlákno – blok – grid [6]

Kernel je funkce, která je prováděna na grafické kartě. Při spuštění vygeneruje velké množství vláken a každé vlákno pak provádí kód kernelu. Pokud jde o zařízení s verzí Compute Capability 1.x , je v jednu chvíli možné mít spuštěn na zařízení jen jeden kernel. Verze 2.x již podporuje více kerenlů spůštěných najednou pro využití veškeré výpočetní kapacity.

Každé vlákno i každý blok má svůj **index** – `threadIdx`, `blockIdx`. Díky těmto indexům se v kernelu zjišťuje, jaké vlákno je zpracováváno. Například `threadIdx.x` nabývá hodnot 0 až `blockDim.x` (počet vláken v bloku pro osu x), `blockIdx.x` nabývá hodnot 0 až `gridDim.x` (počet bloků v gridu pro osu x).



Obrázek 4 Určení vlákna pomocí threadIdx a blockIdx [9]

Funkce vytvářené pro CUDA mohou mít kvalifikátor, určující, pro jaké zařízení bude funkce vytvořena:

- `__global__` - spouštěna z CPU, vykonávána na GPU, musí vracet void, např. kernel()
- `__device__` - volaná z jiné GPU funkce, vykonávána na GPU, nemůže být volána z CPU
- `__host__` - spouštěna z CPU, vykonávána na CPU, např. main()

Pokud funkce nemá žádný kvalifikátor, pak je to automaticky host funkcí, kvůli zpětné kompatibilitě. Pokud totiž chceme do staršího C kódu přidat výpočet kernelem, stávající deklarace funkcí se nemusí přepisovat.

Příklad kernelu pro inkrementaci obsahu matice `matrix` s rozměry `dimx*dimy`:

```
__global__ void matrixIncKernel( int *matrix, int width, int heigth )
{
int ix = blockIdx.x * blockDim.x + threadIdx.x;
int iy = blockIdx.y * blockDim.y + threadIdx.y;
int idx = iy * width + ix;
matrix[idx] = matrix[idx] + 1;
}
```

Obsluha kernelu v kódu CPU:

```
float* deviceMatrix, hostMatrix;
int width, heigth = 128;
int size = width * heigth * sizeof (float);
```

```
cudaMalloc ((void **)& deviceMatrix, size); // alokace paměti na zařízení
cudaMemset (deviceMatrix, 0, size);
dim3 dimBlock (width, heigth); // počet vláken v bloku - 128 x 128
dim3 dimGrid (1,1); // počet bloků v gridu - 1 x 1
// zavolání kernelu
matrixIncKernel <<< dimGrid, dimBlock >>>(deviceMatrix,width,heigth);
hostMatrix = (float *) malloc(size); // alokace paměti na hostu
// zkopírování vypočtené matice z GPU na CPU
cudaMemcpy (hostMatrix, deviceMatrix, size, cudaMemcpyDeviceToHost);
cudaFree(deviceMatrix); // uvolnění paměti na zařízení
free(hostMatrix); // uvolnění paměti na hostu
```

3.2 OpenCL

OpenCL je standardizované API založené na jazyce C pro paralelní programování. Je navrženo tak, aby umožnilo vývoj platformě nezávislých paralelních aplikací. Vývoj OpenCL byl motivován potřebou standardu pro vývoj vysokovýkonných aplikací na stále přibývajícím počtu paralelně-výpočetních platforem. Věnuje se zejména odstranění problémů předchozího programovacích modelů pro paralelně-výpočetní systémy. Vývoj OpenCL byl zahájen firmou Apple, jehož pokračovatelem je nyní Khronos Group, stejná skupina, která vyvíjí standard OpenGL.

OpenCL má složitější model správy zařízení, který odráží jeho přenositelnost mezi různými platformami různých výrobců . Přenositelná OpenCL aplikace musí být připravena na specifika různého hardware, a proto jsou tyto aplikace zpravidla robustnější. Dále je mnoho OpenCL funkcí volitelných a nemusí být podporovány na všech zařízeních. Proto pokud je aplikace zamýšlena být opravdu přenositelná, je potřeba se vyvarovat použití těchto volitelných funkcí. Některé z těchto volitelných funkcí ovšem umožňují dosažení mnohem vyššího výkonu na zařízeních, která je podporují. Ve důsledku tedy nemusí být schopen OpenCL kód dosáhnout zamýšlené výkonnosti na jakémkoli zařízení. [13]

3.2.1 Datový model

Aplikace OpenCL se skládá ze dvou částí – kernelu, vykonávaného na jednom či více OpenCL zařízení a CPU programu, který se stará o spuštění kernelu. Tedy stejně jako u CUDA, způsob, jak spustit paralelní výpočet v OpenCL, je zavolání kernelu ze sekvenčního CPU kódu.

OpenCL terminologie	Ekvivalent v CUDA
Kernel	Kernel
Host program	Host program
NDRange	Grid
Work item (položka)	Thread (vlákno)
Work group (skupina)	Block (blok)

Tabulka 2 Rozdílná terminologie OpenCL a CUDA s totožným významem

Každá položka (work item) má svůj unikátní globální index. Je zde rozdíl mezi CUDA a OpenCL v přístupu k tomuto indexu. **V** CUDA má každé vlákno hodnotu blockIdx a hodnotu threadIdx. Tyto hodnoty se musí zkombinovat, abychom dostali unikátní indexu vlákna. Například pokud má CUDA konfiguraci gridu a bloků jako 2D pole, pak unkátní globální index vlákna v ose x lze získat vyhodnocením b1ockIdx.x * b1ockDim.x + threadIdx.x. V OpenCL kernelu se takovýto index získá zavoláním API funkce get_global_id() s parametrem udávajícím osu - 0 pro x, 1 pro y, 2 pro z.

OpenCL API funkce	Popis	Ekvivalent v CUDA
get_global_id(0);	Globální index položky v ose x	blockIdx.x * blockDim.x + threadIdx.x
get_local_id(0)	Lokální index v rámci skupiny v ose x	threadIdx.x
get_global_size(0);	Velikost NDRange (počet skupin) v ose x	gridDim.x * blockDim.x
get_local_size(0);	Size of each work group in the x-dimension	blockDim.x

Tabulka 3 Rozdíly OpenCL a CUDA v získání indexů a rozměrů

3.2.2 Architektura zařízení

Hostem je tradičně procesor CPU, na kterém se spouští hlavní program. Každé zařízení se seskládá z jedné nebo více výpočetních jednotek (*compute unit* - CU), které odpovídají streaming multiprocesorům (SM) v CUDA. Avšak CU může také odpovídat jádrům CPU nebo jiným typů výpočetních jednotek jako je DSP a FPGA.

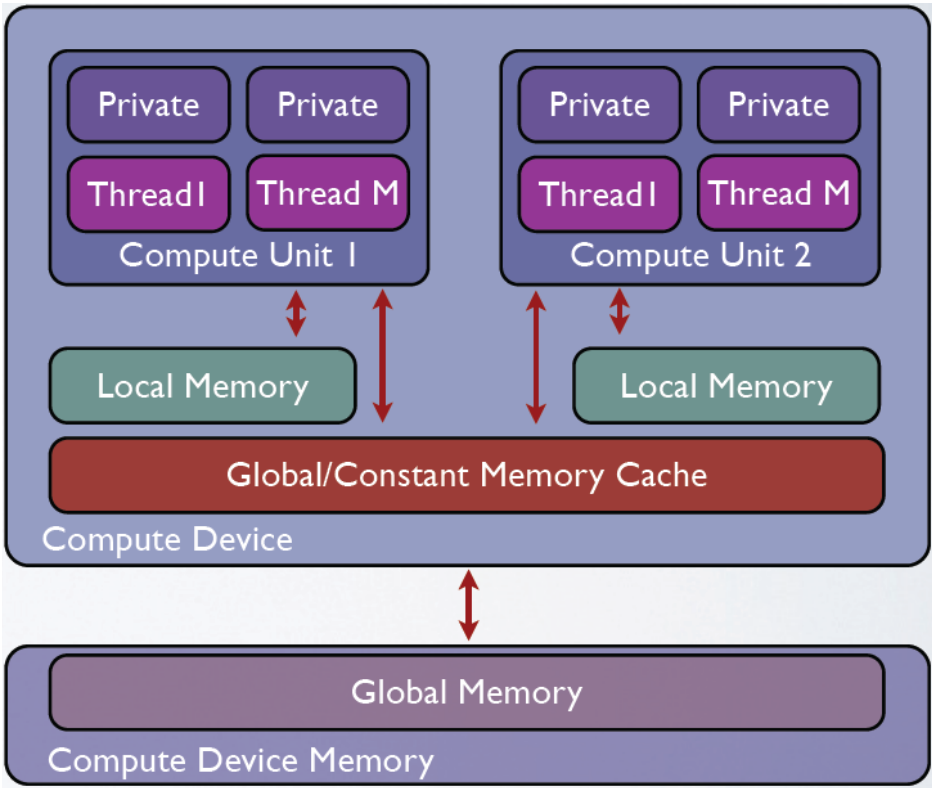
Každá CU sestává z jedné nebo více jednotek pro zpracovávání (processing elements - PE), které odpovídají streamovacím procesorům (SP) v CUDA.

3.2.3 Paměťový model

Všechny OpenCL typy pamětí jsou obdobně čteny/zapisovány jako u CUDA. Velikost konstantní paměti není omezená na 64 kB v OpenCL. Dotaz na zařízení API funkcí (clGetDeviceInfo) vrací podporovanou velikost konstantní paměti.

OpenCL typ paměti	Ekvivalent v CUDA
Globální paměť	Globální paměť
Konstantní paměť	Konstantní paměť
Lokální paměť	Sdílená paměť
Privátní paměť	Lokální paměť

Tabulka 4 Rozdíly terminologie pro typy paměti



Obrázek 5 Paměťový model OpenCL. Převzato z [15]

Funkce clEnqueueReadBuffer() kopíruje data ze zařízení na hosta.

The clCreateBu ffe r funkce odpovídá funkci cud aMalloc() v CUDA.

3.2.4 Kernel

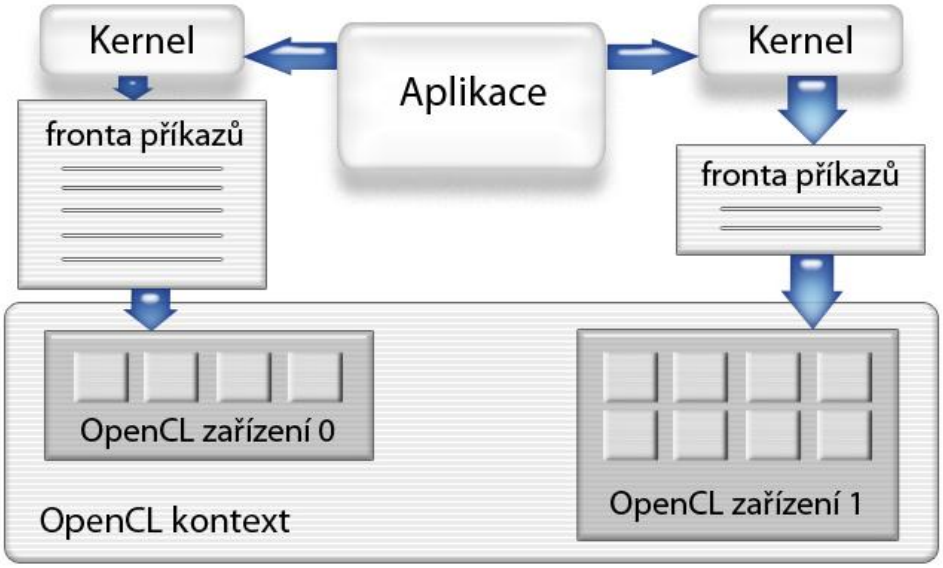
OpenCL kernely mají identickou strukturu jako CUDA kernely. Všechny deklarace OpenCL kernelů začínájí klíčovým slovem __kernel, které je ekvivalentem __globa1__ v CUDA. V CPU kódu je potřeba kernel nahrát do znakového řetězce. Buď se napíše funkce pro převedení obsahu souboru na řetězec a použije se tedy samostaný soubor pro kernel nebo je vložen přímo jako string do kódu CPU. To je ovšem celkem nepraktické, je potom potřeba psát na každý řádek kernelu uvozovací a ukončovací uvozovky a tudíž je takový kernel špatně editovatelný.

Příklad:

```
cl_program program;
cl_kernel kernel;
program = clCreateProgramWithSource(context,1, (const char**)&program_source, NULL,
&err);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "cpotential", &err);
```

3.2.5 Správa zařízení

V nastavování zařízení, která se budou používat pro výpočet, je OpenCL mnohem složitější než CUDA. Tato zvýšená složitost je dána díky podpoře spousty hardwarových platforem.



Obrázek 6 Znázornění hierarchie kontextu a fronty

Všechna dostupná OpenCL zařízení jsou dostupná přes kontext. Pro nastavení jednoho nebo více zařízení v systému je nutné, aby programátor prvně vytvořil kontext, který bude obsahovat tato zařízení. To lze udělat zavoláním OpenCL API funkcí `clCreateContext()` nebo `clCreateContextFromType()`. V aplikaci je typicky potřeba použít funkci `clGetDeviceIDs()` počtu a typů dostupných zařízení a předat výsledek funkcím createContext.

Aby zařízení přijalo nějakou práci pro vykonání, je nutné nejdřív vytvořit frontu příkazů. Pokud je zařízení víc, pak pro každé jednu frontu. Vytvořit frontu lze funkcí clCreateCommandQueue(). Jakmile je již fronta vytvořená, je do ní možné vložit kernel spolu s jeho konfigurací. Zařízení vykonává příkazy od nejstaršího principem LIFO.

Příklad – vytvoření kontextu a fronty:

```
cl_context context;
cl_command_queue cmd_queue;
cl_device_id devices;
clGetDeviceIDs(NULL,CL_DEVICE_TYPE_GPU, 1, &devices, NULL);
context = clCreateContext(0, 1, &devices, NULL, NULL, &err);
cmd_queue = clCreateCommandQueue(context, devices, 0, &err);
```

CUDA tyto složitá nastavení dokáže svými funkcemi schovat pro častý případ, kdy je použito pouze jedno zařízení pro výpočet. Vývojář, který by chtěl mít přímý přístup na všechna CUDA zařízení v systému ovšem může využít nízkoúrovňového CUDA Driver API, kde se použije podobná sekvence příkazů jako v OpenCL ve výchozí podobě.

3.2.6 Spuštění kernelu

V CUDA je kernel spouštěn se syntaxí klasické C funkce s rozšířením o << >>. V OpenCL není žádná přímá funkce na spuštění kernelu. Nejdřív je potřeba přidat argumenty kernelu jeden po druhém funkcí c1SetKerne1Arg(). Následně se musí kernel vložit do fronty funkcí c1EnqueueNDRangeKerne1(). Mezi její parametry je při volání potřeba uvést počet spouštěných položek (work item) a skupin (work group). Kernel se provede, jakmile nebou před ním ve frontě již žádné příkazy.

3.3 DirectCompute

Directcompute je programovací API, které je podmnožinou Microsoft DirectX API. Výhodou je, že podporuje vícero výrobců čipů. DirectCompute podporují některé čipy podporující DirectX 10 a všechny čipy podporují DirectX 11. DirectCompute je zvlášť vhodné pro operace s datovými typy pro foto-video. Kernely – v terminologii DirectCompute „compute shadery“ - se píší v jazyce HLSL a překládají se pomocí fxc DirectX kompileru až při spouštění, stejně jako u OpenCL. Syntaxe jazyka HLSL (High Level Shader Language) je velice podobná C/C++. Možnost použití je omezená na operační systémy Windows Vista SP2, Windows Server 2008, Windows 7 a Server 2008R2. Zájemce odkáži na stránku http://www.danielmoth.com/Blog/DirectCompute.aspx.

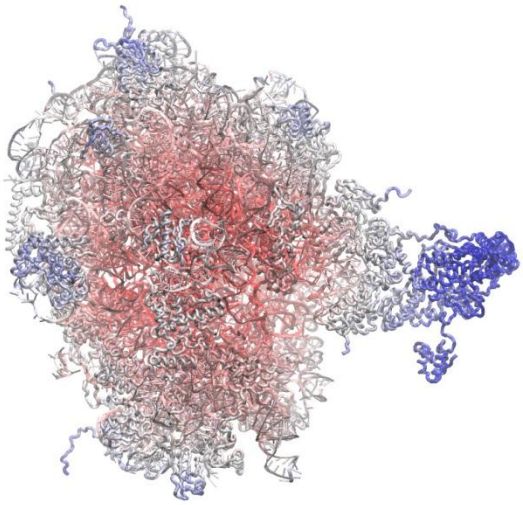
Příklad HLSL kernelu:

- Invertuje barvy 3-kanálové obrázku
- Jeden thread počítá jeden pixel
- Vláken ve skupině (bloku) je 20x20x1
- Pro zpracování obrázku 640x480 je tedy potřeba 32x24 skupin

```
Texture2D<float3>      InputMap : register( t0 );
RWTexture2D<float3>    OutputMap : register( u0 );

[numthreads(20, 20, 1)] // vláken ve skupině bude 20x20x1

void CSMAIN( uint3 GroupID : SV_GroupID, uint3 DispatchThreadID :
SV_DispatchThreadID, uint3 GroupThreadID : SV_GroupThreadID, uint
GroupIndex : SV_GroupIndex )
{
  int3 texturelocation = int3( 0, 0, 0 );
  texturelocation.x = GroupID.x * size_x + GroupThreadID.x;
  texturelocation.y = GroupID.y * size_y + GroupThreadID.y;
  float3 Color = InputMap.Load(texturelocation);

  OutputMap[texturelocation.xy] = float3( 1.0f, 1.0f, 1.0f ) - Color;
}
```

4 Implementace

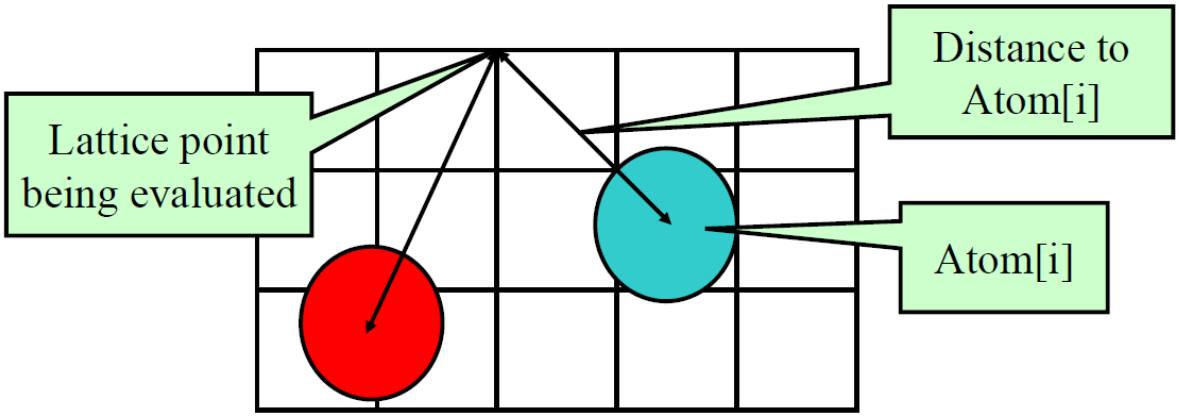
Mým cílem je vytvořit dvě ekvivalentní aplikace, napsané pomocí CUDA a OpenCL, jejichž rychlost výpočtů budu porovnávat. Dále porovnám výhody a nevýhody impementace, na které narazím. Rozhodl jsem se pro vytvoření aplikace počítající elektrický potenciál v krystalické mřížce. Krystalická mřížka je modelem uspořádání atomů v krystalických pevných látkách.



Obrázek 7 Barevně znázorněné velikosti elektrického potenciálu na modelu molekuly. Převzato z [2]

Jako algoritmus jsem zvolil přímý Coulombův součet. Jeho základní myšlenkou je, že pro elektrický potenciál každého bodu krystalické mřížky je součet potenciálů od všech atomů v mřížce. Jeho výhodou je, že pro jeho implementaci není potřeba znát do hloubky problematiku molekul, jako pro jiné algoritmy a je snadně paralelizovatelný pro výpočet na GPU. Důvodem je provádění stejného výpočtu pro všechny body mřížky, jen s jinými daty a fakt, že vlastnosti atomů v mřížce se nemění a pouze se čtou, tudíž se dají nahrát do cachované konstantní paměti.

Cílem aplikace je tedy vypočítat elektrický potenciál pro každý bod krystalické mřížky, který je součtem potenciálů od všech atomů v mřížce, viz Obrázek 8. Při zachování počtu atomů na určitou vzdálenost se zvyšuje s rosoucí mřížkou i počet atomů v mřížce. Počet výpočtů tedy roste kvadraticky. Algoritmus má proto kvadratickou časovou složitost - O(M*N) pro M bodů mřížky a N atomů v mřížce - a není vhodný pro vypočty velkých rozměrů mřížky (cca nad 1+e6 Å; jeden Ångström [Å] má délku 0,1 nm).

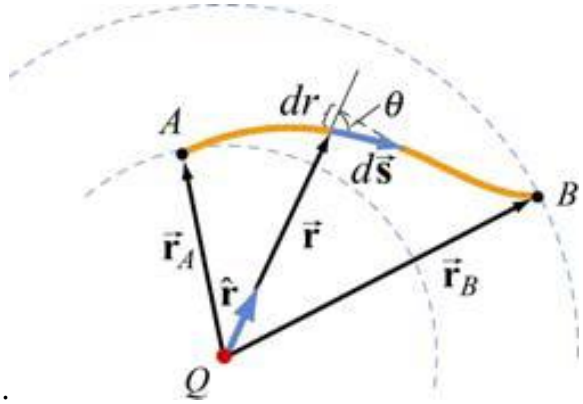


Obrázek 8 Pro každý bod mřížky jsou připočteny potenciály od všech atomů v mřížce
© John E. Stone, 2007, University of Illinois
http://web.mit.edu/8.02t/www/802TEAL3D/visualizations/coursenotes/modules/guide03.pdf

4.1 Odvození vzorce výpočtu

Elektrický (nebo též elektrostatický) potenciál je veličina, která popisuje potenciální energii jednotkového elektrického náboje,. tzn. množství práce potřebné pro přenesení jednotkového elektrického náboje ze vztažného bodu, kterému je přisouzen nulový potenciál, do daného místa. Za vztažný bod se obvykle bere nekonečně vzdálený bod, daleko od vlivu jiných nábojů. [http://cs.wikipedia.org/wiki/Elektrick%C3%BD_potenci%C3%A1l]

Elektrické pole produkované nábojem Q je $\vec{E} = (\frac{Q}{4\pi\varepsilon_0 r^2})\hat{r}$, kde $\hat{r}$ je jednotkový vektor směřující k bodům A a B.



Obrázek 9 Rozdíl potenciálu mezi dvěma body A a B vlivem náboje Q

Z Obrázku 9 je vidět, že $\hat{r} \cdot d\vec{s} = ds \cdot cos\theta = dr$, z čehož vyplývá

$$\Delta V = V_B - V_A = -\int_A^B \frac{Q}{4\pi\varepsilon_0 r^2} dr = \frac{Q}{4\pi\varepsilon_0}\left(\frac{1}{r_B} - \frac{1}{r_A}\right)$$

V praxi se často jako vztažný bod volí nekonečno, takže elektrický potenciál v bodě P je

$$V_P = -\int_\infty^P \vec{E} \cdot d\vec{s}$$

Elektrický potenciál ve vzdálenosti r od náboje Q je tedy

$$V(r) = \frac{1}{4\pi\varepsilon_0}\frac{Q}{r}$$

Jakmile je přítomen více než jeden atom, pak použitím principu superpozice získáme electrický potenciál jako součet potenciálů vzniklých vlivem jednotlivých atomů.

$$V(r) = \frac{1}{4\pi\varepsilon_0}\sum_i \frac{q_i}{r_i} = k_e \sum_i \frac{q_i}{r_i}$$

4.2 Vytvoření kernelu

Konstantní paměť GPU není velká (64 KB) a může do ní být zapisováno pouze v kódu CPU a tak je potřeba zavolat kernel několikrát, aby se sečetly všechny příspěvky potenciálů od všech atomů. V praxi se do konstantní paměti vejde přes 4000 atomů a s tolika je tedy spouštěn každý kernel (kromě posledního, kde může být jakkoliv velký zbytek menší než 4000 atomů). Pro jeden atom jsou ukládány souřadnice x, y, z v mřížce a náboj, celkem tedy 4 float proměnné na atom. Do konstantní paměti se vejde tedy $65536\ B\ /\ (4\ *4\ B)\ =\ 4096$ atomů.

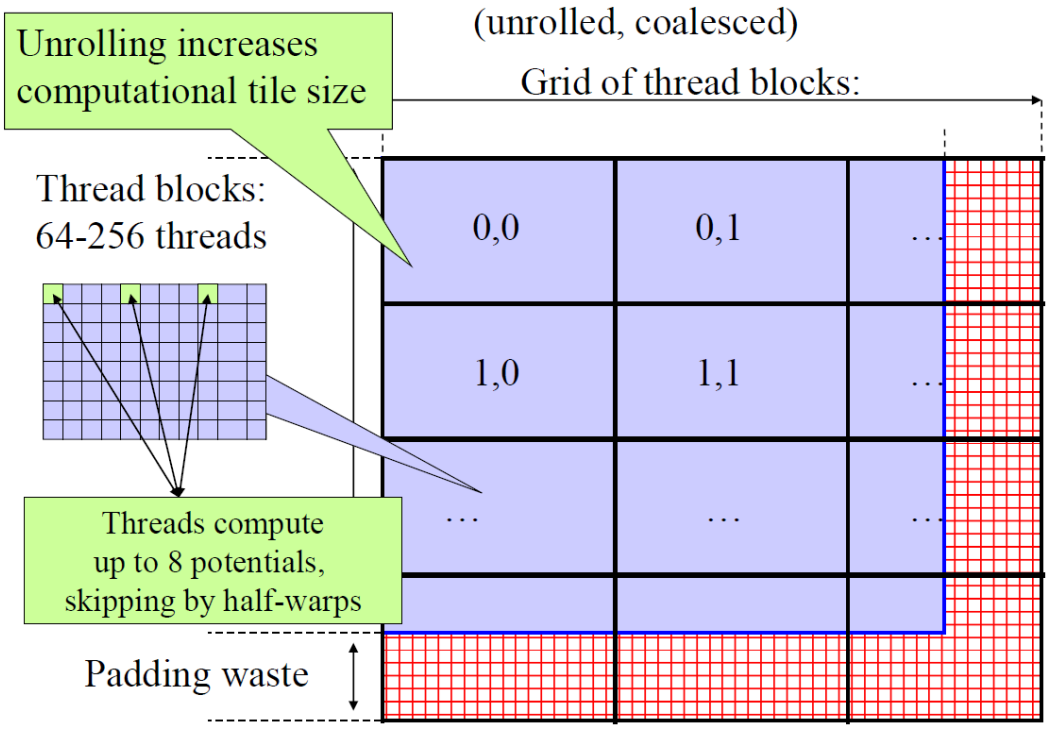
Pro zjištění, kolik je obvykle obsaženo atomů v mřížce na jednotku délky, můžeme vycházet například z křemíku. Základem krystalické mřížky je elementární buňka - rovnoběžnostěn o hraně *a* (mřížkový parametr) – která se v mřížce periodicky opakuje. Křemík má mřížkový parametr rovný 5,43 Å, tedy buňka má obsah $a^3 = 5{,}43^3 = 160{,}1$ Å^3. Do této elementární buňky se vejde 8 atomů, protože jde o diamantovou strukturu. Tedy počet atomů v křemíku:

$$\frac{8 \text{ Si atomů}}{160{,}1 \text{ Å}^3} = 0{,}05 \text{ atomů na Å}^3$$

To odpovídá jednomu atomu na 20 Ångströmů krychlových. [11]

Nejvíce časově náročnou operací ve vláknu je odmocnina, kterou bohužel nelze eliminovat.

Krystalová mřížka je tří dimenzionální objekt, který lze jednoduše zpracovávat po vrstvách jako 2D pole. Tato 2D zpracová kernel vždy po částech v blocích. Pro urychlení je v jednom vlákně načítáno osm po sobě následujících paměťových míst – bodů mřížky.



Obrázek 10 Dekompozice mapy potenciálu – jedné vrstvy - do CUDA bloků. Pževzato z [14]

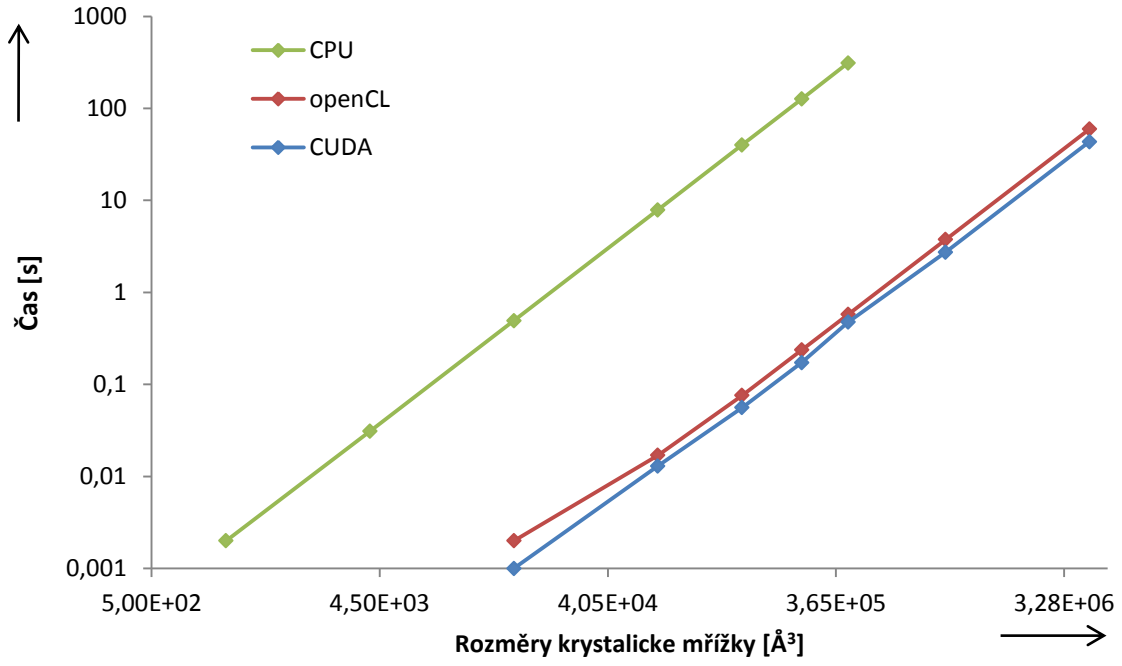
5 Výkonnostní testy

Testoval jsem CUDA i OpenCL verzi aplikace na grafické kartě NVIDIA GeForce GTX285, procesoru Intel(R) Core(TM) i7 CPU 920 @ 2.67GHz, OS Ubuntu 9.1 x64. Obě knihovny CUDA i OpenCL byly ve verzi 2.3.

Protože kernely v OpenCL a CUDA jsou velice podobné, tak rozdíly ve výkonnosti mohou být přisouzeny efektivitě jednotlivých knihoven.

Aplikace má tyto kroky:

- Nastavení GPU – zahrnuje detekci GPU, kompilování kernel OpenCL, atd
- Inicializace atomů – náhodné generování jejich umístění v mřížce a náboje
- Zkopírování časti atomů na GPU do konstantní paměti
- Spuštění po sobě několika kernelů na GPU, celkově zpracující všechny atomy
- Zkopírování vypočítaných elektrických polí zpět na CPU
- Výpis času potřebného pro výpočet

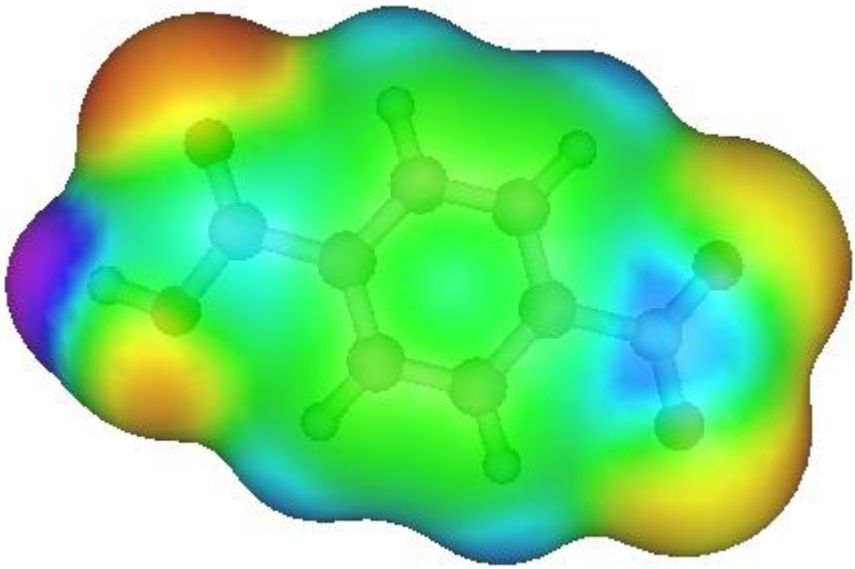


Graf 3 Graf rychlosti výpočtu v závislosti na velikosti problému

Velikost mřížky [Å^3]	Počet atomů v mřížce	Doba výpočtu CPU [ms]	Doba výpočtu CUDA [ms]	Zrychlení CUDA oproti CPU	Doba výpočtu OpenCL [ms]	OpenCL pomalejší než CUDA
1024	51	2	0	2x	0	-
4096	204	31	0	31x	0	-
16384	819	493	1	493x	2	-
65536	3276	7895	13	607x	17	1,31x
147456	7372	39993	56	714x	76	1,36x
262144	13107	126267	173	730x	237	1,37x
409600	20480	311274	476	654x	576	1,21x
1048576	52428	-	2739	-	3761	1,37x
4194304	209715	-	43370		59606	1,37x

Tabulka 5 Porovnání počtu atomů s velikostí mřížky

Z tabulky je vidět, že výpočet problému na CPU je neúnosně se prodlužující s velikostí mřížky a počtem atomů.



Obrázek 11 Barevné vyobrazené hodnot elektrických potenciálů. Převzato z [12]

6 Závěr

Oba programovací modely mají podobnou funkcionalitu. Přepis kódu z CUDA do OpenCL se zdá teoreticky jako jednoduchý úkol, ovšem v rozdílech terminologie se dá lehce ztratit. Změny v kernelu jsou opravdu minimální, ovšem kód v OpenCL pro CPU, kde se nastavuje zařízení a nahrání kernelu mi přišlo zbytečně zdlouhavé a neintuitivní, je nutné ho napsat nový.

Z výsledků vyplývá, že CUDA přináší vyšší výkon na stejné aplikaci, až 1,37x. Další výhodou CUDA je, že je programovatelná na lehce vyšší úrovni abstrakce a není potřeba tolik úsilí k vytvoření aplikace. Pro použití OpenCL hovoří přístup „napiš jednou, otestuj všude“, kdy je přenositelná mezi různými výpočetními platformami, byť pro každé zařízení je vhodné optimalizovat kód tak, aby se dosáhlo potenciálně nejvyšší rychlosti.

Konkrétní implementace výpočtu elektrického potenciálu v krystalové mřížce je využitelná například pro určení polohy ion v struktuře molekuly pro molekulární dynamickou simulaci, kde je spolu s elektrickým potenciálem nutné počítat ještě další fyzikální rovnice.

Literatura

[1] MAHAPATRA N.; VENKATRAO B.: *The Processor-Memory bottleneck: Problems and Solutions*, [cit. 2010-05-19]. Dostupné z WWW: <http://www.acm.org/crossroads/xrds5-3/pmgap.html>.

[2] Visual Molecular Dynamics, University Of Illinois Of Urbana, [cit. 2010-05-19]. Dostupné z WWW: <http://www.ks.uiuc.edu/Research/vmd/vmd-1.8.6/>.

[3] OWENS, J.: *Parallel Architectures*, Lecture 20, UC Davis, [cit. 2010-05-19]. Dostupné z WWW: <http://www.nvidia.com/content/cudazone/cudau/courses/ucdavis/lectures/wrapup.pdf>.

[4] ŠTEFEK, P.: *Ian Buck z Nvidie o budoucnosti GPGPU*, Svět hardware, [cit. 2010-05-19]. Dostupné z WWW: <http://www.svethardware.cz/art_doc-12683E4FB834D7E1C1257631005DD96D.html>.

[5] ZAORÁLEK L.: *Úvod do technologie CUDA: Hello CUDA!,* Root.cz, [cit. 2010-05-19]. Dostupné z WWW: <http://www.root.cz/clanky/uvod-do-technologie-cuda-hello-cuda/>.

[6] NVIDIA CUDA Programming Guide, NVIDIA, [cit. 2010-05-19]. Dostupné z WWW: <NVIDIA_CUDA_Programming_Guide_2.3.pdf>.

[7] NVIDIA CUDA C Programming Best Practices Guide, NVIDIA, [cit. 2010-05-19]. Dostupné z WWW: <NVIDIA_CUDA_BestPracticesGuide_2.3.pdf>.

[8] NVIDIA CUDA Architecture, NVIDIA, [cit. 2010-05-19]. Dostupné z WWW: <http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf>.

[9] BUCK, I.: *Programming CUDA*, NVIDIA, [cit. 2010-05-19]. Dostupné z WWW: <SC07_CUDA_2_Language_Buck.pdf>.

[10] RABAEY, J.: *Low Power Design Essentials*, Springer, 2009

[11] CROUSE, D.: *Homework 1 Grading Key*, City College of New York [cit. 2010-05-19]. Dostupné z WWW: <http://www-ee.ccny.cuny.edu/www/web/crouse/EE339/Homework/Homework%201%20Solutions%20Fall03.htm>.

[12] BOTTYAN, T.: *Electrostatic Potential Maps*, UC Davis, [cit. 2010-05-19]. Dostupné z WWW: <http://chemwiki.ucdavis.edu/Theoretical_Chemistry/Chemical_Bonding/Electrostatic_Potential_maps>.

[13] HWU, W.; KIRK, D.: *Programming Massively Parallel Processors: A Hands-on Approach*, Elsevier Science, 2010

[14] STONE J.: *Applied parallel programming*, Lecture 19, University of Illinois, Urbana-Champaign, 2007

[15] Gohara, D.: *Episode 2 - OpenCL Fundamentals, Washington University School of Medicine, St. Louis*, [cit. 2010-05-19]. Dostupné z WWW: <http://www.macresearch.org/files/opencl/Episode_2.pdf>

Dodatek A

Technická vybavenost zařízení NVIDIA (Compute Capability)

Technickou vybavenost a vlastnosti potřebné pro programování zařízení vyjadřuje tzv. Compute Capability. Exitují různé verze, určující například počet registrů, počet procesorů nebo maximální počet vláken spustitelných na procesoru.

Compute Capability je definováno hlavním číslem verze, za tečkou následuje vedlejší číslo verze. Zařízení se stejným hlavním číslem verze vychází ze stejné architektury jádra. Vedlejší číslo verze odpovídá určitému zlepšení architektury jádra a novým vlastnostem. Například zařízení s kódovým označením GeForce, Quadro a Tesla mají verzi 1.x, zařízení architektury Fermi má verzi 2.0

<table>
<tr><th></th><th colspan="5">Compute Capability</th></tr>
<tr><th>Vlastnosti</th><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.0</th></tr>
<tr><td>Double-precision floating-point numbers</td><td colspan="3">No</td><td colspan="2">Yes</td></tr>
<tr><td>Maximum x- or y-dimension of a grid of thread blocks</td><td colspan="5">65535</td></tr>
<tr><td>Maximum number of threads per block</td><td colspan="4" rowspan="2">512</td><td rowspan="2">1024</td></tr>
<tr><td>Maximum x- or y-dimension of a block</td></tr>
<tr><td>Maximum z-dimension of a block</td><td colspan="5">64</td></tr>
<tr><td>Threads in a warp</td><td colspan="5">32</td></tr>
<tr><td>Maximum number of blocks per multiprocessor</td><td colspan="5">8</td></tr>
<tr><td>Maximum number of warps per multiprocessor</td><td colspan="2">24</td><td colspan="2">32</td><td>48</td></tr>
<tr><td>Maximum number of threads per multiprocessor</td><td colspan="2">768</td><td colspan="2">1024</td><td>1536</td></tr>
<tr><td>Number of 32-bit registers per multiprocessor</td><td colspan="2">8 K</td><td colspan="2">16 K</td><td>32 K</td></tr>
<tr><td>Maximum amount of shared memory per multiprocessor</td><td colspan="4">16 KB</td><td>48 KB</td></tr>
<tr><td>Amount of local memory per thread</td><td colspan="4">16 KB</td><td>512 KB</td></tr>
<tr><td>Constant memory size</td><td colspan="5">64 KB</td></tr>
<tr><td>Cache working set per multiprocessor for constant memory</td><td colspan="5">8 KB</td></tr>
<tr><td>Maximum number of instructions per kernel</td><td colspan="5">2 million</td></tr>
</table>