VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

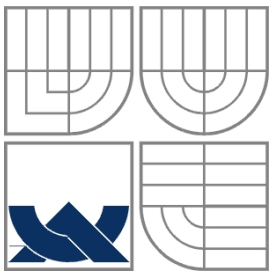
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

VYUŽITÍ GRAFICKÉHO ADAPTÉRU PRO OBECNÉ VÝPOČTY

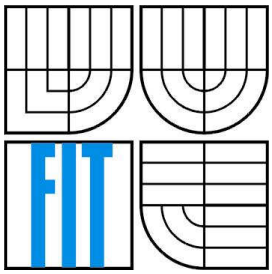
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE VLADIMÍR KRÉZEK
AUTHOR

BRNO 2009



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 BASED ON GRAPHICS PROCESSING UNITS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

VLADIMÍR KRÉZEK

VEDOUCÍ PRÁCE
SUPERVISOR

ING. JIŘÍ JAROŠ

BRNO 2009

Abstrakt

Bakalářská práce pojednává o využití grafického adaptéru (GPU) pro obecné výpočty jinak zpracovávané na procesoru (CPU). Zabývá se knihovnami, které pro tento účel lze využít a to především architekturou nVidia CUDA.

Abstract

This bachelor thesis discusses the usage of a graphic adapter (GPU) for general computation otherwise processed on a processor (CPU). It deals with libraries that can be used for this purpose, especially with CUDA architecture.

Klíčová slova

GPGPU, CUDA, ATI Stream, EcoLib, StreamIt, OpenCL, RapidMind, Gauss-Seidelova metoda.

Keywords

GPGPU, CUDA, ATI Stream, EcoLib, StreamIt, OpenCL, RapidMind, Gauss-Seidel method.

Citace

Vladimír Krézek: Využití grafického adaptéru pro obecné výpočty, bakalářská práce, Brno, FIT VUT v Brně, 2009

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.

……
Vladimír Krézek
18.5.2009

Poděkování

Děkuji svému vedoucímu za jeho rady a připomínky. Dále bych chtěl také poděkovat rodině a přátelům, kteří mě podporovali.

Obsah

1 Úvod

Příchodem grafických karet podporující rozhraní Microsoft DirectX 8, obsahující shadery (programovatelné jednotky pro vykonání programů přímo na grafické kartě), se nově objevuje možnost využití tohoto adaptéru pro obecné výpočty. Tomuto způsobu využití se říká GPGPU (General-Purpose Computing on Grafics Processing Units), neboli volně řečeno obecné výpočty prováděné na grafickém procesoru (GPU, Grafic Processing Unit). Termín GPGPU (spolu s ním i vznikla internetová stránka gpgpu.org) byl definován již v roce 2002 Markem Harrisem [1].

Tato práce se zabývá právě tímto způsobem programování. Na úvod zde bude uvedeno několik základních informací o této technice:

- Proč právě grafická karta?
- Historie grafických karet
- Výkonnost výpočtů na GPU
- Výhody vs. nevýhody

Další kapitoly se průběžně budou zabývat knihovnami, které lze použít. Architektuře CUDA od firmy nVidia, na kterou jsem se v této práci zaměřil, je vyhrazena zvláštní kapitola. Zbylé kapitoly se věnují návrhu a implementací příkladů. V závěru jsou zhodnoceny dosažené výsledky a diskutovány možnosti využití v praxi.

1.1 Proč právě grafická karta?

Do doby než se objevilo GPGPU, byli programátoři zvyklí programovat na klasických procesorech (CPU, Central Processing Unit). Kromě procesorů se v počítačích objevuje ještě jedna výkonná výpočetní jednotka – grafický procesor, který je přímo stavěný pro zpracování velkého množství dat. Ve skutečnosti byl vyvinut výhradně pro paralelní zpracování velkého množství pixelů. Paralelismus je zajištěn multiprocesory umístěnými na grafickém čipu, které jsou navíc velice výkonné. Využívá se technika SIMD (Single Instruction Multiple Data), tzn. že na větším počtu dat se vykoná právě jedna instrukce. Naproti tomu klasický procesor využívá SISD (Single Instruction Single Data), což znamená, že nad daty je postupně vykonávána instrukce – v podstatě odpovídá von Neumannově architektuře. Zde je nutné podotknout, že GPU sice obsahuje několik výpočetních jednotek (jader), na druhou stranu dokáže pracovat pouze se stejným typem dat (viz obrázek 1.1). Z konstatovaného vyplívá, že GPGPU se hodí právě pro výpočty jako jsou např. náročné operace nad velkým množstvím dat, různé simulace nebo dokonce i vykonávání náročných dotazů nad databází. Toto vše však ještě neznamená, že GPU by měl nahradit CPU. Cílem je, aby se GPU využil jako koprocesor/akcelerátor ke klasickému procesoru, ne však aby jej zcela nahradil.

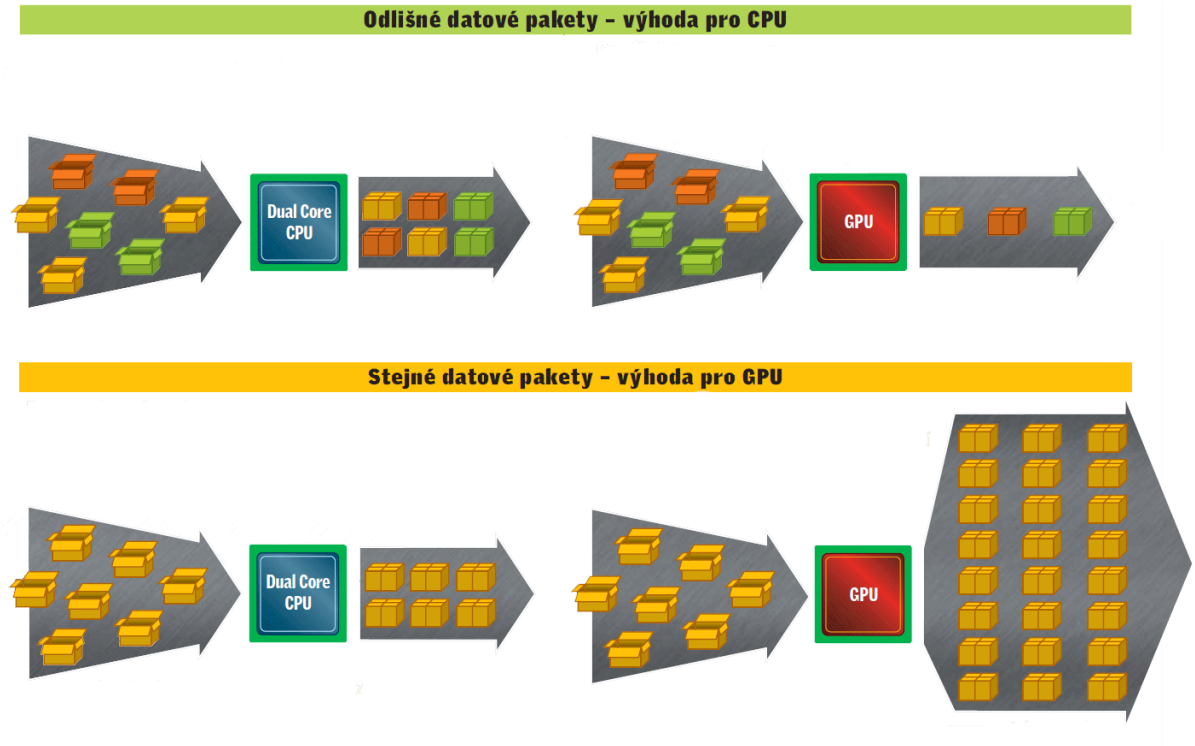
1.2 Historie GPU

Z vývojového hlediska můžeme grafické karty rozdělit do pěti generací. První generaci některá rozdělení ignorují nebo ji uvádějí jako „předgenerační“ grafické karty [5]. Ty jsou integrovány do systémové desky včetně video paměti.

Druhá generace se může vztahovat k roku 1998, kdy se objevují i první grafické karty s AGP portem. Hlavními představiteli jsou Riva TNT od firmy nVidia a 3dfx Voodoo3. Tyto procesory převážně prováděly rasterizaci, která však vyžadoval již transformované vrcholy. Tuto operaci v této generaci stále ještě prováděl CPU. Některé čipy podporovali „multitexturing“, který byl schopen

v rasterizačním kroku míchat většinou dvě, ale i více textur dohromady. Tyto čipy mohly v sobě obsahovat až deset milionů transistorů.

Na přelomu roku 1999 se objevily grafické procesory třetí generace, které díky provádění transformace a osvětlení přímo na čipu bývají také nazývány T&L GPU (tedy transformation and lighting). Právě tyto výpočty dříve vykonávané klasickými procesory byly kvůli frekventovanosti přesunuty již na stranu grafické karty. Díky tomuto, dostává programátor do rukou nové operace pro programování, které však ve skutečnosti ještě nejsou plně programovatelné. Zástupci této generace mohou být jmenovány nVidia Geforce256 nebo GeForce 2 a z dílny ATI Radeon 7500, kteří obsahují v řádu 25 milionů transistorů.



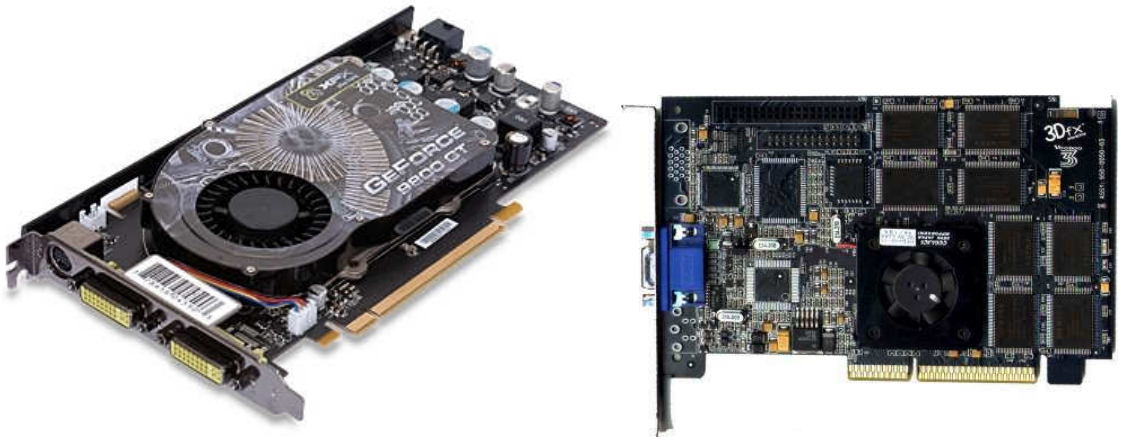
Obr. 1.1 Zpracování dat na CPU a GPU [2]

Další generace (konec roku 2001) zaznamenává významnou změnu, která v budoucnu umožní využít zřetězeného zpracování (pipelining) na grafickém hardwaru podobně jako na streamovacím procesoru. V této generaci se poprvé objevuje programovatelná jednotka neboli shader, čímž se také poprvé nakročuje k možnosti vykonávat na grafické kartě obecné výpočty. Jedná se pouze o programovatelnou jednotku pro výpočty na vrcholech (vertex shader) a o velmi omezenou jednotku pro výpočty na úrovni pixelů (pixel/fragment shader). Omezení spočívá ve způsobu přístupu k texturám, v omezení formátu dat a také díky absenci řízení kontroly toku programu (větvení). Dalším vylepšením těchto čipů může být uvedena nativní podpora stínování a 3D textur. Produkty této generace mohou obsahovat okolo 60 milionů tranzistorů. Jako zástupci mohou být jmenovány nVidia GeForce 3 případně GeForce 4 a od konkurence ATI Radeon 8500.

Poslední celkem dlouhá a zároveň ne nejmladší generace v podstatě vystihuje současnou situaci. V této chvíli se již shadery zobecňují a stávají se plně programovatelnými. A to jak vertex, tak i pixel shadery. Generace začíná sérií GeForce FX od firmy nVidia a Radonem 9700 od ATI (dnes již patří pod značku AMD), přičemž každá v sobě obsahuje již více jak sto milionů tranzistorů.

V této době se dokonce objevují i grafické karty zaměřené přímo na obecné výpočty. Je jím řešení od firmy nVidia a její Tesla, která svým výkonem (přes 500 GFLOP* [7] na jeden GPU) může konkurovat i některým superpočítačům. Existuje několik verzí a to jak pro stolní počítače, tak i jako

clusterové systémy. K tomuto hardwaru je navíc od stejné firmy nabízena technologie CUDA (viz další kapitoly).



Obr. 1.2 Grafické karty nVidia GeForce 9800GT (vlevo) a 3dfx VooDoo3 (vpravo)

1.3 Výkonnost GPU

V roce 1965 Gordon Moore předpověděl, že počet tranzistorů, který může být umístěn na jednom čipu, bude každým rokem dvojnásobný [4]. Jeho předpověď trvá dodnes a zdá se, že ještě nějaký ten rok bude platit. Během čtyřiceti let, kdy byl Moorův zákon vyřčen se počet tranzistorů zvedl z padesáti až na neuvěřitelné stovky milionů.

Nové čipy však nenavyšují pouze počet tranzistorů, ale navíc také zmenšují jejich velikost, přičemž díky menší velikosti mohou pracovat rychleji než jejich předchůdci. Zvýšení rychlosti tranzistoru se odráží ve zvýšení hodinového kmitočtu. Dohromady se zvýšením počtu tranzistorů a rychlostí hodin se výkon procesoru zvýší okolo 71% ročně.

Polovodičová paměť, která používá jemně odlišnou technologii, také těží z podobných výrobních pokroků. Předpovídá se, že kapacita paměti (DRAM) se každé tři roky zdvojnásobí. Výkon paměti se dá měřit ze dvou hledisek: šířka pásma, což je počet dat, které se přesunou každou sekundu nebo latenci, což je doba než dostaneme odpověď na dotazovaná data. Nicméně výkon pamětí neroste, tak rychle jako výkon procesorů.

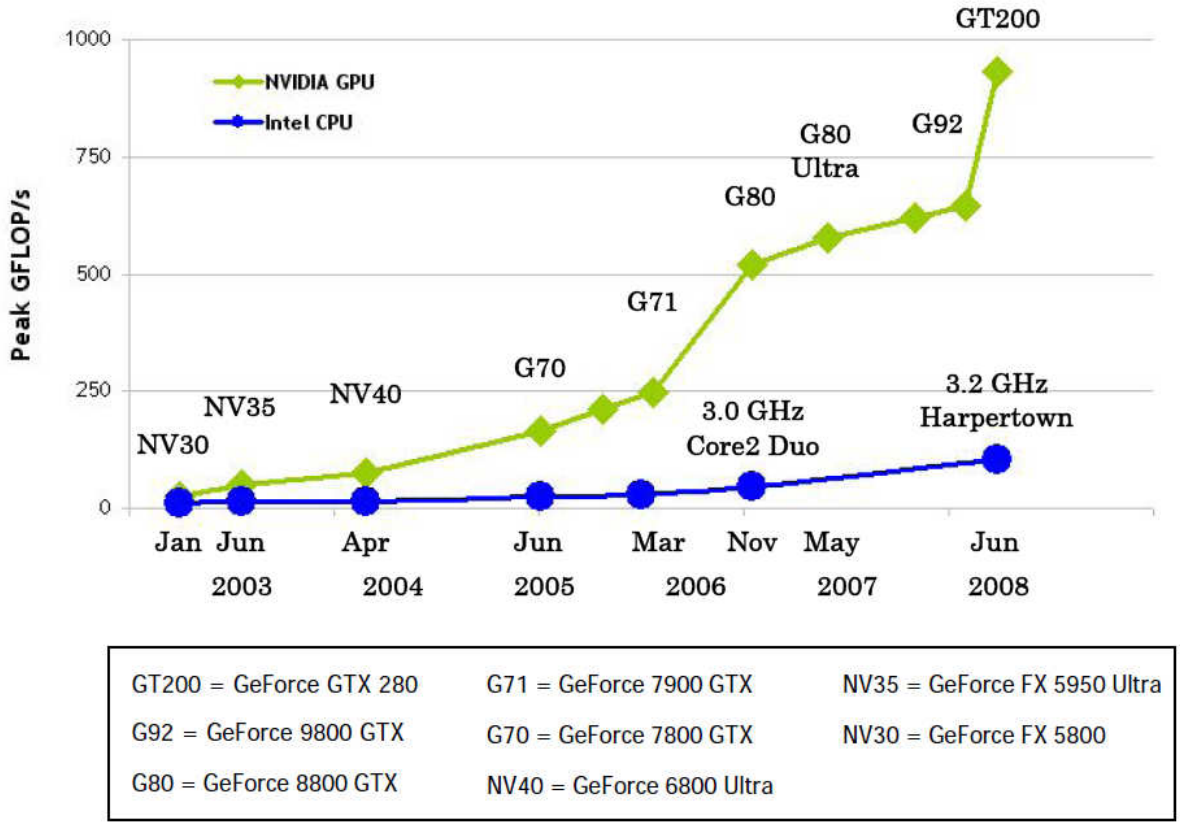
Obecně předchozí věty můžeme brát jako pozitiva, problém nastává v tom, že u procesorů se zvyšuje výkon rychleji než u pamětí, což způsobuje určité mezery mezi výpočetní silou a komunikací mezi procesorem a pamětí.

Navíc předchozí odstavce se věnovaly spíše CPU, než grafický procesorům. U grafických procesorů vývoj probíhá ještě rychleji (např. v dnešní době je situace následující: hrubý výkon CPU Intel Core i7-965 je 51,2 GFLOP [17], oproti tomu hrubý výkon GPU nVidia GTX280 je neuvěřitelných 933 GFLOP [6]) a to díky neustálému nátlaku ze strany jak uživatelů, tak i ze strany např. herního průmyslu, který vyžaduje co nejrealističtější zobrazení v počítačových hrách. Nicméně i tak se společnosti zabývající grafickým hardwarem mohou pochlubit svým ročním obratem, i když spoustu financí investují do neustálého vývoje.

Rychlost hodin, tak i velikost čipu se zvětšuje. Čas, který zabere cesta signálu přes celý čip, měřený v hodinových cyklech, se také zvětšuje. Na dnešních nejrychlejších procesorech posílání signálu z jedná stany čipu na druhou typicky vyžaduje více hodinových cyklů a tento čas vzrůstá

* 1 GFLOP = jedna miliarda operaci v plovoucí řádové čárce za sekundu

s každou další generací. Tento trend můžeme charakterizovat jako nárůst ceny komunikace v porovnání s cenou výpočtu. Důsledkem toho budou v budoucnu návrháři více využívat výpočtů pomocí tranzistorů, aby odstranili potřebu drahé komunikace. Další pravděpodobný dopad bude růst počtu výpočtů použitelných na slovo z šířky pásma paměti. Příkladem porovnejme tři ve své době vlajkové lodě nVidie (r. 2002 – GeForce FX 5800, r. 2003 – GeForce FX5950 a r. 2004 GeForce 6800). Zatímco GeForce FX 5800 mohla připustit dvě operace v plovoucí řádové čárce na každé slovo šířky pásma, GeForce FX 5950 mohla připustit 2,66 operací a GeForce 6800 téměř 6 operací. Předpokládá se, že tento trend bude pokračovat i v příštích generacích.



Obr. 1.3 Vývoj výkonuGPU oproti CPU[8]

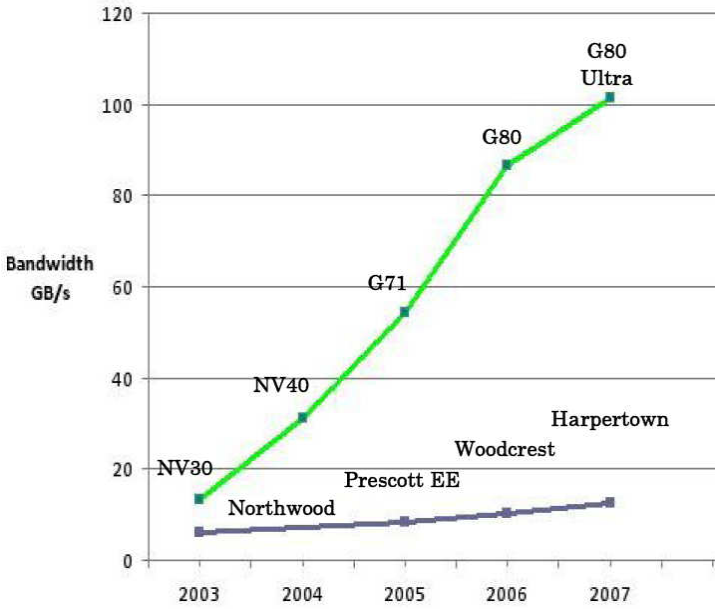
Mezera mezi šířkou pásma a latencí bude také důležitým prvkem budoucích architektur. Protože latence se zlepšuje mnohem pomaleji než šířka pásma, musí návrháři implementovat řešení, která do doby než budou vrácena potřebná data (což v některých případech může trvat dlouhou dobu) se bude vykonávat nějaká užitečná práce.

I když menší tranzistory vyžadují méně energie, nárůst počtu tranzistorů na procesoru je vyšší než snižování potřebné energie. Proto každá generace potřebuje čím dál více energie (grafická karta řady nVidia GTX 200 v nejhorším případě vyžaduje až 226 W [6].

Abychom využili bohatého výkonu, který nabízejí dnešní grafické karty, a dosáhli tak vysokého výkonu našich aplikací, je potřebami si uvědomit dvě věci: za prvé je potřeba rozprostřít výpočetní sílu tak, aby výkon hardwaru byl co nejvíce využit a za druhé je potřeba zajistit efektivní komunikaci v podobě dobře uspořádaných dat. Efektivita komunikace je potřeba hlavně proto, aby výpočetní zdroje byly dobře zásobovány dostatkem dat.

1.4 Výhody vs. nevýhody

V předcházející podkapitole bylo hovořeno o vysokém výkonu grafických karet, což můžeme brát jako největší výhodu tohoto způsobu programování. Navíc si musíme uvědomit, že všechny současné počítače disponují grafickou kartou. Z toho důvodu je jasné, že přístup k tomuto má v podstatě každý. To, že některé knihovny vyžadují specifický hardware, to už je ovšem věc jiná. Navíc ceny grafických karet např. oproti superpočítačům jsou velmi nízké.



Obr. 1.4 Vývoj šířky pásma GPU oproti CPU [8]

Existují zde však i určitá negativa: ne každý grafický čip je vhodný pro jakoukoliv knihovnu (viz CUDA pro nVidie a ATI/AMD Stream pro stejnojmenné grafické karty) nebo některé knihovny nejsou lehce dostupné (např. RapidMind nebo EcoLib jen jako trial verze). Pro některé programátory však může být největším problémem přechod na tento programovací model, který se od klasického modelu liší. Navíc je nutné programovat přímo v assembleru dané architektury nebo ve specializovaném jazyce (který je většinou podobný jazyku C). Další a asi ne posledním problémem může být to, že se architektura tohoto programování mění tak rychle, že může docházet ke změnám úplných základů knihoven (množství knihoven jsou teprve v beta- nebo dokonce i alfa-testování).

2 Knihovny pro GPGPU

Tato kapitola se věnuje knihovnám, které lze využít pro obecné výpočty na GPU. Začneme od základních jazyků, které jsou nativně určeny pro programování grafiky (proto jsem od programování s pomocí těchto jazyků ustoupil) a jen díky tomu, že podporují programování shaderů je lze využít právě pro GPGPU. Mezi tyto jazyky, které vycházejí z jazyka C patří GLSL (OpenGL Shading Language jako součástí OpenGL) a HLSL (High Level Shading Language jako součástí DirectX), který byl vyvíjen po boku jazyka Cg (od společnosti nVidia) a který mu je velice podobný. Cg [9] neboli C for Graphics historicky vychází z několika modelů. Vychází z jazyka C, ze kterého zdědil jak syntaxi, tak i sémantiku, a spojuje v sobě koncepty, které jsou určeny pro non-realtime shading jazyky jako je RenderMan (API vyvinuté animačním studiem Pixar pro foto realistické zobrazování, využíváno i Hollywoodem) a svoji funkcionalitu zakládá na rozhraní OpenGL a Direct3D pro vykreslování 3D real-time grafiky.

Poté, co se zjistilo, že grafické adaptéry lze využít i jinak než pro zobrazování, začaly vznikat knihovny na bázi jazyka C/C++ přímo stavěné pro GPGPU. Mezi ně patří hlavně knihovny vyprodukované výrobci grafických karet – nVidia s architekturou CUDA a ATI Stream (dnes již pod značkou AMD). Tyto knihovny však jsou použitelné pouze na hardwaru vyprodukovaném těmito výrobci.

ATI Stream SDK [11], dříve známo pod názvem CTM (Close To Metal), zahrnuje jednak překladač Brook+, což je hardwarově optimalizovaná verze překladače vyvinutého na Stanfordské univerzitě, který je optimalizovaný pro stream computing a jednak CAL (Compute Abstraction Layer), který je určen pro nízkoúrovňové programování. Jeho výhodou je stejně jako u ostatních knihoven datový paralelismus, který dovoluje paralelní vykonaní operace nad větším objemem dat, a síla aritmetických výpočtů. Architektura staví na proudech dat (stream), které se vykonávají přes jádra (kernel) na grafickém adaptéru. Klasické C rozšiřuje o definování právě těchto streamů a speciální označování funkcí pro rozlišení kernelů. Pro běh aplikací vyvinuté pod touto knihovnou je zapotřebí minimálně ATI Radeon HD2350. Knihovna se může pochlubit i svým komerčním využitím. Video editor PowerDirector 7 zpřístupnil rozšířenou verzi o HW akceleraci enkódování pomocí ATI Stream. Toto rozšíření však neexistuje pouze pro ATI stream, ale i pro CUDA, kde dokonce dosahuje lepších výsledků a více možností. Využití zasáhlo také herní průmysl, kde se ve hře Froblins snaží urychlovat a vylepšovat umělou inteligenci. Následuje příklad jednoduchého kernelu, který sečte dvě matice a následně je zavolán z funkce main():

```
// kernel, který sečte matice A a B a výsledek uloží do streamu
// result
kernel void addMatrix(double A<>, double B<>, out double result<>) {
     result = A + B;
}

int main(int argc, char** argv) {
     // vytvoření streamů
     double A<Height, Width>;
     double B<Height, Width>;
     double C<Height, Width>;

     // zápis dat do streamů
     streamRead(A, inputA);
     streamRead(B, inputB);

     // zavolání kernelu
```

```
    addMatrix(A, B, C);

    // zápis dat zpět ze streamu
    streamWrite(C, output);
}
```

Funkce `addMatrix` je definicí jádra, které jako parametry přijímá streamy. Jimi jsou dva vstupní streamy `A` a `B` a jeden výstupní `result`. Kernel vykonává pouze sečtení těchto streamů a vložení výsledku na výstupní stream. Ve funkci `main` se vytvoří patřičné streamy tak, že za datový typ a název streamu se do špičatých závorek vloží jeho velikost. Pomocí funkce `streamRead` proběhne namapování vstupních matic na vstupní streamy, zavolá se jádro a výsledný stream se funkcí `streamWrite` zapíše do matice.

RapidMind, EcoLib, StreamIt a OpenCL jsou dalšími knihovnami, které mohou být zmíněny. Některé z nich však nejsou volně dostupné, případně jsou ve svých počátcích.

RapidMind [12] (celý název zní RapidMind Multi-core Development Platform) je produkt stejnojmenné kanadské firmy, který minimálně a intuitivně rozšiřuje C++. Lze jej využít nejen pro klasické a grafické procesory, ale i pro procesory IBM cell (http://www.research.ibm.com/cell/). Rozdíly mezi těmito architekturami RapidMind úplně stírá, tzn. není třeba psát jednotlivé verze pro každou architekturu. Obsahuje tzv. Code Optimizer, který analyzuje a optimalizuje paralelizaci výpočtů, Load Balancer, který plánuje a synchronizuje práci tak, aby všechna jádra byla co nejlépe využita, Data Manager, který se automaticky stará o přenos dat v rámci paměti (což je důležité hlavně u grafické paměti, která je úzkým hrdlem celého výpočetního systému) a nakonec nástroj pro zaznamenávání dosaženého výkonu.

V příkladu, který dále následuje, je opět popsáno sčítání matic. Klíčová slova `BEGIN` a `END` znázorňují začátek a konec kernelu, který obsahuje definici vstupních streamů (`In<Value3f> in1, in2;`), které se sečtou a jejichž výsledek je uložen do jednoho výstupního steamu (`Out<Value3f> out;`). `Value3f` znázorňuje datový typ, jimž v tomto případě je vektor, který obsahuje tři čísla datového typu `float`. Ve funkci `main` pak kromě inicializace (`Rapidmind::init();`) proběhne vytvoření pole dvou streamů stejného typu jako v definici kernelu a přiřazení výsledku výpočtu do proměnné `a`.

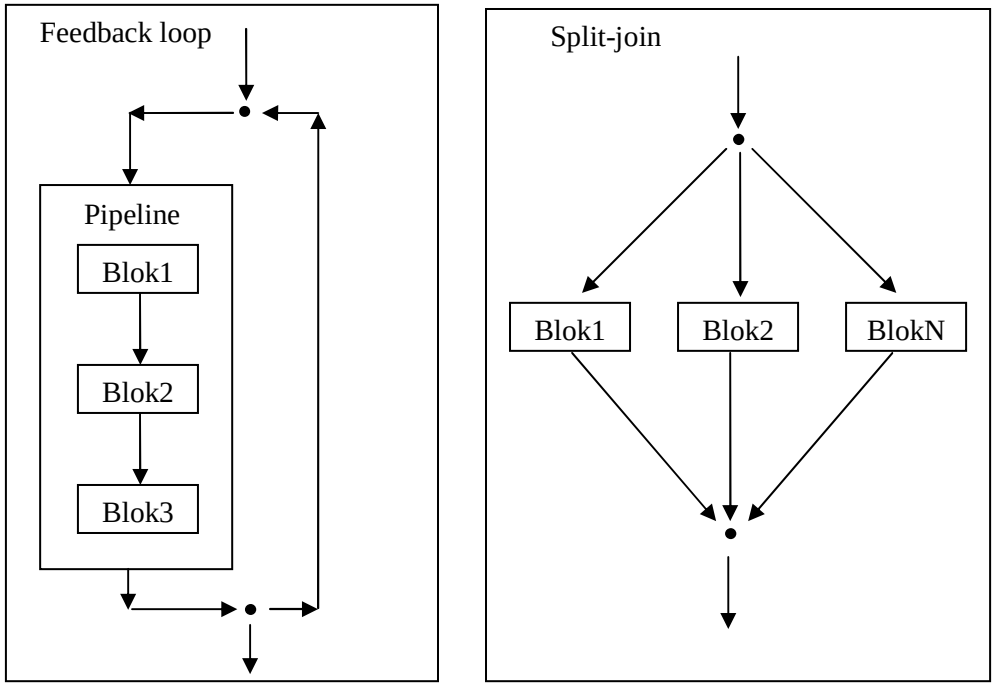
Příklad [15] použití knihovny RapidMind:

```
// kernel, který provádí sčítání matic
Program addMatrix = BEGIN {
    In<Value3f> in1, in2;
    Out<Value3f> out;

    out = in1 + in2;
} END;

int main(int argc, char** argv){
    Rapidmind::init();

    Array<2, Value3f> a(512, 512), b(512, 512);
    a = addMatrix(a, b);
}
```

Další knihovnou, která podporuje běh na grafických kartách AMD i nVidia, které mají implementováné pixel shadery od verze 2.0 je knihovna EcoLib [13]. Pro její používání lze využít C++ nebo Javu a je nutné mít nainstalovaný DirectX 9.0c. Obsahuje dalších pět knihoven, které slouží pro generování náhodných čísel, počítání lineární, vektorové nebo maticové algebry (BLAS),

řešení lineárních rovnic (LAPAC) nebo knihovnu pro zpracování signálů. Zabezpečuje optimalizovaný přenos dat mezi procesorem a grafickou kartou. Po vyplnění internetového formuláře by na Vámi zadanou emailovou adresu měl přijít odkaz na stažení trial verze této knihovny. Mně však ani po třech týdnech čekání, žádný odkaz ani zpráva nebyla doručena.

Zajímavou knihovnou je StremIt (projekt MIT) [14], která je zamýšlená pro zjednodušení výpočtů se signály a pro další proudové výpočty. Programátor si vytvoří blokové schéma obsahující elementární bloky zvané filtry, které mají jeden vstup a jeden výstup a které se dále spojují a vytvářejí tak celou strukturu schématu. Existují tři druhy těchto spojení (viz obrázek): jednoduché spojení filtrů za sebou (pipeline), větvení filtrů (split-join) a zpětná vazba (feedback loop). Tato spojení mají stejně jako filtry jeden vstup a jeden výstup. Jako programovací jazyk je v tomto případě zvolena Java, jejíž kompilátor spolu s touto knihovnou vygeneruje efektivní kód určený pro cílovou architekturu.



Obr. 2.1 Znázornění schémat používaná v StreamIt

Následující příklad použití StreamIt počítá průměr 10 vstupních čísel, které začínají nulou a inkrementálně zvyšují svou hodnotu o 1 až do hodnoty devět. Program je rozdělen do tří filtrů (v kódu funkce s klíčovým slovem `filter`), které se sériově (`pipeline`) propojí ve funkci `Minimal` (`add` přidá daný filtr do řady). Před názvem a typem funkce se vždy definuje datový typ vstupního a výstupního proudu (např. `void->int`). Filtr `IntSource` ve své inicializační části `init` nastaví proměnnou `x` na počáteční hodnotu a tu v části `work` vloží do proudu. Filtr `Averager` (počítá průměr) vždy jedno číslo z proudu vybere (funkce `pop`) a jedno číslo do proudu vloží (funkce `pop`) a to přesně `n`-krát. Ve for-cyklu je ukázáno, jak lze přečíst (funkce `peek`) číslo z proudu. Poslední filtr `IntPrinter` načte číslo z proudu a vypíše jej na standardní výstup.

Příklad použití knihovny StreamIt:

```
// vytvoření pipeline
void->void pipeline Minimal {
      add IntSource();
      add Averager(10);
      add IntPrinter();
}

// vstupní stream
void->int filter IntSource {
      int x;
      init { x = 0; }
      work push 1 { push(x++) };
}

// výpočet průměru
int->int filter Averager(int n) {
      work pop 1 push 1 peek n {
            int sum = 0;
            for (int i = 0; i < n; i++)
                  sum += peek(i);
            push(sum/n);
            pop();
      }
}

// výstupní stream
int->void filter IntPrinter {
      work pop 1 { print(pop()); }
}
```

OpenCL (Open Computin Language) [15] je první otevřený standard pro programování paralelních heterogenních systémů pro obecné výpočty určený pro vícejádrové CPU, GPU, Cell procesory a další. Podporuje vývoj širokého spektra aplikací, od vestavěného softwaru, přes uživatelské aplikace až k HPC (High Performance Compution) řešením. Byl vytvořen skupinou Khronos za účasti mnoha průmyslových společností jako je AMD, Apple, Broadcom, IBM, Intel, Motorola, Nokia, nVidia, RapidMind a dalších. Je to vcelku nový standart, který je v době psaní této práce ještě v plenkách, proto zde není uvedeno více informací.

3 Architektura CUDA

V listopadu roku 2006 nVidia představila Compute Unified Device Architecture (CUDA) [8], paralelní architekturu pro obecné výpočty s novým paralelním programovacím modelem a instrukční sadou, která povyšuje engine grafických karet nVidie pro řešení mnoha komplexních výpočetních problémů a to efektivnější cestou než na CPU.

CUDA přichází se softwarovým řešením, které dovoluje vývojářům použít vysokoúrovňový jazyk C. Programátoři však nejsou omezeni pouze na tento jazyk, ale lze využít např. i Fortran nebo C++.

Programovací model je vytvořen pro ulehčení programování tak, aby skryl paralelismus. Jádrem jsou tři klíčové abstrakce – hierarchické skupiny vláken, sdílená paměť a bariérová synchronizace – které jsou programátorovi předloženy v minimálních rozšířeních jazyka. Navíc je tu výběr ze dvou způsobů implementace. A to buď pomocí runtime knihovny (zvoleno pro tuto práci), která je postavena na druhém způsobu a tím je nízko úrovňové Driver API.

CUDA aplikace je možné vyvíjet jak na nejvýkonnějších grafických kartách jako je GeForce GTX280, případně na profesionálních produktech známé pod názvem Tesla, tak i na levných a lehce dostupných adaptérech (GeForce 8100 a vyšší, Quadro Plex 1000 a vyšší, Quadro NVS 130 a vyšší a Quadro FX 360 a vyšší).

Jazyk C pro CUDA rozšiřuje jeho základy, aby vývojář mohl definovat funkce, které se budou vykonávat na GPU. Takováto funkce se nazývá jádro (kernel). Kernel je definován pomocí klíčového slova `__global__` a při jeho volání se udává počet vláken. Každé vlákno je rozeznáno svým unikátním identifikátorem, který je zpřístupňován pomocí vestavěné proměnné `threadIdx`. Pro úplnost, `threadIdx` je třídimenzionální vektor, takže vlákna mohou být identifikována použitím jedno-, dvou- nebo třírozměrným indexem. Tyto indexy pak vcelku vytváří blok vláken. Toto poskytuje přirozenou cestu, jak vyvolat výpočet přes elementy domény jako je vektor nebo matice.

Vlákna v rámci bloku mohou mezi sebou sdílet data s použitím sdílené paměti. Pro synchronizaci je pak nutně do určitého místa v kódu vložit bariéru `__syncthreads()`, která zajistí, aby další kód nebyl vykonán až do té doby než nad všemi vlákny se vykonají definované operace. Z toho důvodu je počet vláken v bloku omezen. V dnešních GPU jsou vlákna omezeny na 512. Nicméně jádro může běžet nad více bloky, takže počet vláken může být navýšen. Tyto bloky pak mohou být organizovány do jednorozměrné nebo dvourozměrné mřížky, kde jsou bloky identifikovány pomocí indexu `blockIdx`. Proměnná `blockDim` vyjadřuje počet právě těchto bloků v mřížce.

Bloky je nezbytné vykonávat nezávisle, aby je bylo možné vykonávat v jakémkoliv pořadí ať už paralelně nebo sériově.

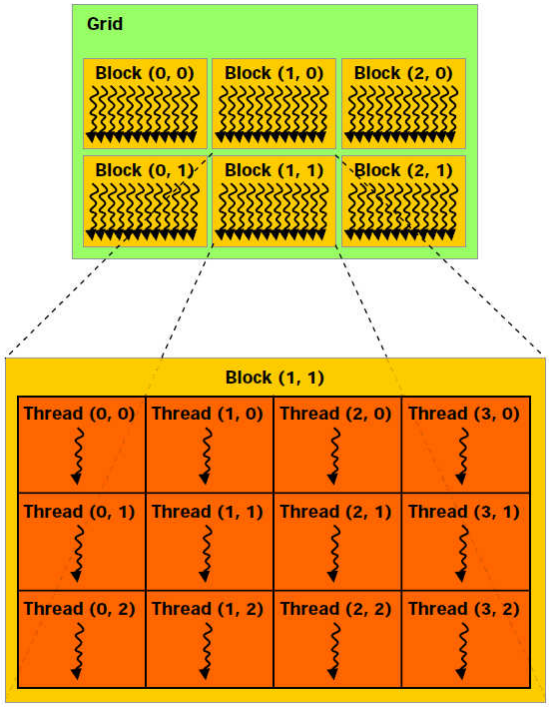
Každé vlákno má svou lokální paměť, každý blok má sdílenou paměť, ke které mají přístup vlákna tohoto bloku. Navíc, všechny vlákna mohou přistupovat ke globální paměti. Nakonec zde máme konstantní paměť a paměť pro textury, které slouží pouze pro čtení.

Výpočetní způsobilost je definována hlavním a vedlejším číslem revize. Hlavní číslo vypovídá o architektuře jádra, zatímco vedlejší revizní číslo označuje vylepšení, která se projevují nad danou architekturou. V příloze A je výčet dosud existujících způsobilostí.

Aplikace, ve kterých byla CUDA využita, jsou např.:

- Knihovna pro Matlab, díky které kód Matlabu běží na GPU
- Power Direcotr, jenž byl zmíněn již v předchozí kapitole
- Seti@home – vědecká oblast, která má za cíl detekovat inteligentní život mimo Zemi
- Folding@home – vědecká oblast, která na základě proteinů zkoumá nemoci

- Kódování videa (např. TMPGEnc 4.0 Xpress, Badaboom Media Converter)
- Geografický informační systém od firmy Manifold
- Elomsoft Distributed Password Recovery – aplikace pro nalezení hesla od souborů produktu MS Office, PDF, ZIP/RAR archívy a další
- Pluginy pro Photoshop.



Obr. 3.1 Uspořádání vláken do bloků a mřížky [8]

V příloze B je předveden příklad sčítaní matic s využitím této knihovny.

4 Návrh a implementace příkladů

Implementovat jsem se rozhodl výpočet soustavy *n* rovnic o *n* neznámých pomocí Gauss-Seidelovy iterační metody a pro druhý příklad byla zvolena simulace pohybu částic. Knihovnu pro implementaci jsem vybíral z dvojice ATI Stream a nVidia CUDA. Nakonec jsem se rozhodl pro architekturu CUDA, protože počítač, na kterém probíhal vývoj těchto aplikací, disponuje grafickou kartou nVidia GeForce 9400GT, dále také kvůli tomu, že ATI Stream je stále ve fázi beta testování.

4.1 Gauss-Seidelova metoda

Gauss-Seidelova metoda [10] patří mezi iterační metody, které na rozdíl od metod přímých, po předem daném konečném počtu kroků nevedou k přesnému řešení. U iteračních metod se nejprve zvolí počáteční aproximace řešení a určitým postupem ji v každém kroku metody zlepšíme. K řešení se postupně přibližujeme a obecně ho dosáhneme až v limitě. Protože výpočet nelze provádět do nekonečna, po určité době výpočet přerušíme (standardně se výpočet ukončuje po dosažení určité velikosti chyby). Výsledkem se tedy stává pouze přibližné řešení soustavy. Mezi tyto metody patří Jacobiho a Gauss-Seidelova. Pro lepší pochopení implementované metody zde bude vysvětlena i metoda Jacobiho.

4.1.1 Jacobiho metoda

Nejprve si popíšeme, jak se Jacobiho metodou soustavy rovnic řeší a poté si tuto metodu vyzkoušíme na příkladu.
Budeme pracovat se soustavou lineárních rovnic:

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ &a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ &\qquad\vdots \\ &a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{aligned} \tag{4.1}$$

Z první rovnice vyjádříme x_1, z druhé rovnice x_2 atd. Dostaneme:

$$\begin{aligned} x_1 &= \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3 - \cdots - a_{1n}x_n) \\ x_2 &= \frac{1}{a_{22}}(b_2 - a_{21}x_1 - a_{23}x_3 - \cdots - a_{2n}x_n) \\ &\vdots \\ x_n &= \frac{1}{a_{nn}}(b_n - a_{n1}x_1 - a_{n2}x_2 - \ldots - a_{nn-1}x_{n-1}) \end{aligned} \tag{4.2}$$

Řešení soustavy budeme hledat následujícím způsobem:
Libovolně zvolíme počáteční aproximaci řešení $x^{(0)} = (x_1^{(0)}, x_2^{(0)}, \cdots, x_n^{(0)})^T$.
Tato čísla dosadíme do pravé strany rovnice 4.2. Tím dostaneme novou aproximaci řešení $x^{(1)} = (x_1^{(1)}, x_2^{(1)}, \cdots, x_n^{(1)})^T$. Tu opět dosadíme do pravé strany 4.2 atd.

Obecně každou další aproximaci řešení získáme podle předpisu:

$$\begin{aligned} x_1^{(r+1)} &= \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(r)} - a_{13}x_3^{(r)} - \cdots - a_{1n}x_n^{(r)}) \\ x_2^{(r+1)} &= \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(r)} - a_{23}x_3^{(r)} - \cdots - a_{2n}x_n^{(r)}) \\ &\vdots \\ x_n^{(r+1)} &= \frac{1}{a_{nn}}(b_n - a_{n1}x_1^{(r)} - a_{n2}x_2^{(r)} - \cdots - a_{nn-1}x_{n-1}^{(r)}) \end{aligned} \qquad (4.3)$$

Za jistých podmínek (dále popsaných) se tímto postupem budeme přibližovat k přesnému řešení soustavy. Ve výpočtu pokračujeme, dokud se nedosáhne určité předem dané přesnosti, např. dokud se aproximace řešení neustálí na požadovaném počtu desetinných míst, nebo dokud není překročen předem daný maximální počet kroků. Jacobiho metodou nemusíme řešení soustavy najít vždy. V některých případech posloupnost postupných aproximací k řešení soustavy nekonverguje. Uvedeme nyní podmínky, které zaručí, že metoda konverguje (tj. najdeme pomocí ní přibližné řešení).

Definice. Matice *A* se nazývá řádkově ostře diagonálně dominantní právě tehdy, když

$$|a_{ii}| > \sum_{j=1, j\neq i}^{n} |a_{ij}| \quad \text{pro } i = 1, \cdots, n \quad (4.4)$$

(neboli když je v každém řádku matice absolutní hodnota prvku na diagonále vetší než součet absolutních hodnot všech ostatních prvku v onom řádku) a sloupcově ostře diagonálně dominantní právě tehdy, když:

$$|a_{jj}| > \sum_{j=1, j\neq i}^{n} |a_{ij}| \quad \text{pro } j = 1, \cdots, n \quad (4.5)$$

(neboli když je v každém sloupci matice absolutní hodnota prvku na diagonále vetší než součet absolutních hodnot všech ostatních prvku v onom sloupci). Je-li matice soustavy ostře řádkově nebo sloupcově diagonálně dominantní, Jacobiho metoda konverguje. Jestliže matice soustavy není diagonálně dominantní, Jacobiho metoda konvergovat muže a nemusí. Existuje podmínka pro konvergenci Jacobiho metody nutná a dostatečná (tj. pokud je splněna, metoda konverguje a pokud není splněna, metoda diverguje), jenže je pro velké matice prakticky neověřitelná. Proto, nejsme-li si jisti konvergencí metody, je vhodné stanovit maximální počet kroku a je-li překročen, výpočet ukončit s tím, že metoda diverguje. Pak je potřeba zvolit jinou metodu nebo soustavu nějak upravit.

Příklad 4.1 Jacobiho metodou řešte soustavu:

$$\begin{aligned} 15x_1 - x_2 + 2x_3 &= 30 \\ 2x_1 - 10x_2 + x_3 &= 23 \\ x_1 + 3x_2 + 18x_3 &= -22 \end{aligned} \qquad (4.6)$$

Řešení: Matice soustavy je diagonálně dominantní, protože platí:

$$|15| > |-1| + |2|, \quad |-10| > |2| + |1|, \quad |18| > |1| + |3|$$

Proto je konvergence metody zaručena. Vypíšeme iterační vztahy:

$$x_1^{(r+1)} = \frac{1}{15}(30 + x_2^{(r)} - 2x_3^{(r)})$$
$$x_2^{(r+1)} = -\frac{1}{10}(23 - 2x_1^{(r)} - x_3^{(r)}) \quad (4.7)$$
$$x_3^{(r+1)} = \frac{1}{18}(-22 - x_1^{(r)} - 3x_2^{(r)})$$

Jako počáteční aproximaci zvolíme $x = (0, 0, 0)^T$. Postupně získávané aproximace řešení budeme zapisovat do tabulky:

r	$x_1^{(r)}$	$x_2^{(r)}$	$x_3^{(r)}$
0	0	0	0
1	2	-2,3	-1,2222
2	2,0096	-2,0222	-0,9500
3	1,9918	-1,9930	-0,9968
4	2,0000	-2,0013	-1,0007

Tabulka 4.1

Je vidět, že posloupnost postupných aproximací konverguje k řešení soustavy (2,-2,-1). Kdybychom chtěli získat řešení s přesností ε = 0, 01, mohli bychom nyní výpočet zastavit, protože:

$$\left| x_1^{(4)} - x_1^{(3)} \right| = \left| 2{,}0000 - 1{,}9918 \right| < 0{,}01$$
$$\left| x_2^{(4)} - x_2^{(3)} \right| = \left| -2{,}0013 - (-1{,}9930) \right| < 0{,}01$$
$$\left| x_3^{(4)} - x_3^{(3)} \right| = \left| -1{,}0007 - (-0{,}9968) \right| < 0{,}01$$

zatímco kdybychom požadovali přesnost ε = 0, 001, museli bychom ve výpočtu pokračovat, protože např. $\left| x_1^{(4)} - x_1^{(3)} \right| > 0{,}001$.

Kdybychom rovnice z předcházejícího příkladu přepsali v jiném pořadí, např.

$$x_1 + 3x_2 + 18x_3 = -22$$
$$15x_1 - x_2 + 2x_3 = 30 \quad (4.8)$$
$$2x_1 - 10x_2 + x_3 = 23$$

příslušné iterační vztahy by vypadaly takto:

$$x_1^{(r+1)} = -22 - 3x_2^{(r)} - 18x_3^{(r)}$$
$$x_2^{(r+1)} = -30 + 15x_1^{(r)} + 2x_3^{(r)} \quad (4.9)$$
$$x_3^{(r+1)} = 23 - 2x_1^{(r)} + 10x_2^{(r)}$$

Podmínka konvergence metody není splněna. Podívejme se, jak se budou chovat postupné aproximace řešení:

r	$x_1^{(r)}$	$x_2^{(r)}$	$x_3^{(r)}$
0	0	0	0
1	-22	-30	23
2	-346	-314	-233
3	5114	-5686	-2425

Tabulka 4.2

Na první pohled je zřejmé, že k řešení soustavy (2,−2,−1) touto cestou nedojdeme, metoda diverguje.

4.1.2 Gauss-Seidelova metoda

Gauss-Seidelova metoda je velmi podobná metodě Jacobiho. Liší se od ní pouze v tom, že při výpočtu další aproximace řešení použijeme vždy nejnovější přibližné hodnoty *x1, x2, … , x_n,* které máme k dispozici. Podrobněji: $x_1^{(r+1)}$ vypočteme stejně jako u Jacobiho metody a při výpočtu $x_2^{(r+1)}$ je ihned použijeme (zatímco u Jacobiho metody jsme použili staré $x_1^{(r)}$). Při výpočtu $x_3^{(r+1)}$ použijeme nové $x_1^{(r+1)}$ a $x_2^{(r+1)}$ atd.
Obecně iterační vztahy vypadají takto:

$$
\begin{aligned}
x_1^{(r+1)} &= \frac{1}{a_{11}}(b_1 - a_{12}x_2^r - a_{13}x_3^{(r)} - \cdots - a_{1n}x_n^{(r)}) \\
x_2^{(r+1)} &= \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(r+1)} - a_{23}x_3^{(r)} - \cdots - a_{2n}x_2^{(r)}) \\
x_3^{(r+1)} &= \frac{1}{a_{33}}(b_3 - a_{31}x_1^{(r+1)} - a_{32}x_2^{(r+1)} - \cdots - a_{3n}x_n^{(r)}) \qquad (4.10) \\
&\vdots \\
x_n^{(r+1)} &= \frac{1}{a_{nn}}(b_n - a_{n1}x_1^{(r+1)} - a_{n2}x_2^{(r+1)} - \cdots - a_{nn-1}x_{n-1}^{(r+1)})
\end{aligned}
$$

Dá se dokázat, že je-li matice soustavy ostře řádkově nebo sloupcově diagonálně dominantní, Gauss-Seidelova metoda konverguje.

Příklad 4.2 Gauss-Seidelovou metodou řešte tutéž soustavu jako v předchozím příkladu t.j.

$$
\begin{aligned}
15x_1 - x_2 + 2x_3 &= 30 \\
2x_1 - 10x_2 + x_3 &= 23 \qquad (4.11) \\
x_1 + 3x_2 + 18x_3 &= -22
\end{aligned}
$$

Řešení: Již jsme ověřili, že podmínka konvergence je splněna. Vypíšeme iterační vztahy:

$$x_1^{(r+1)} = \frac{1}{15}(30 + x_2^{(r)} - 2x_3^{(r)})$$
$$x_2^{(r+1)} = -\frac{1}{10}(23 - 2x_1^{(r+1)} - x_3^{(r)}) \quad (4.12)$$
$$x_3^{(r+1)} = \frac{1}{18}(-22 - x_1^{(r+1)} - 3x_2^{(r+1)})$$

Jako počáteční aproximaci zvolíme opět $x = (0, 0, 0)^T$.

r	$x_1^{(r)}$	$x_2^{(r)}$	$x_3^{(r)}$
0	0	0	0
1	2	-1,9	-1,0167
2	2,0089	-1,9999	-1,0005
3	2,0001	-2,0000	-1,0000
4	2,0000	-2,0000	-1,0000

Tabulka 4.3

Vidíme, že se k řešení soustavy přibližujeme rychleji než pomocí Jacobiho metody. I obecně se dá říci, že Gauss-Seidelova metoda obvykle konverguje rychleji než metoda Jacobiho. Proto se používá častěji.

4.1.3 Implementace

Implementace vychází z předchozích rovnic, protože jejich algoritmizace není nijak složitá:

Algoritmus 4.1
```
if (matice je diagonálně dominantní) {
  do {
    for (i < počet rovnic) {
      for (j < počet rovnic) {
        if (i ≠ j) // pokud prvek není na diagonále
          Spočítej a přičti j-tý parametr levé strany rovnice;
      }
      Přičti pravou stranu rovnice;
      Dosažený i-tý výsledek vyděl prvkem na diagonále;
      Zapiš jako výsledek této aproximace;
      Vypočítej přesnost i-tého výpočtu;
    }
  } while (není dosaženo zadané přesnosti)
}
```

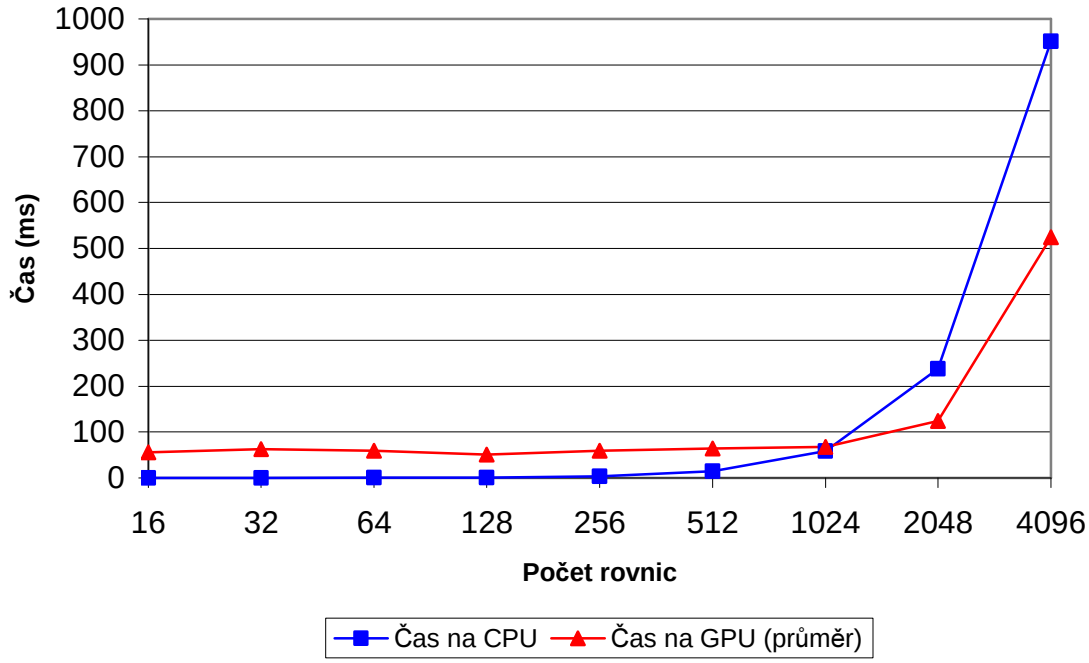
Tento algoritmus vyjadřuje jak by soustava byla vypočtena s pomocí CPU. Abychom algoritmus optimalizovali pro použití na GPU, musíme ho paralelizovat. Toho dosáhneme tak, že každou rovnici přiřadíme jednomu vláknu:

Algoritmus 4.2
```
if (matice je diagonálně dominantní) {
```

```
  do {
      for (i < počet rovnic) {
        if (index vlákna ≠ j) // pokud prvek není na diagonále
          Spočítej a přičti i-tý parametr levé strany rovnice;
      }
      Přičti pravou stranu rovnice;
      Dosažený výsledek aktuálního vlákna vyděl prvkem na diagonále;
      Zapiš jako výsledek této aproximace;
       Vypočítej přesnost tohoto výpočtu;
    }
  } while (není dosaženo zadané přesnosti)
}
```

Tímto nám pak odpadá použití jednoho for-cyklu, místo něhož se právě používají vlákna. Z tohoto pak vyplývá i nastavení mřížky. Tedy budeme pracovat s jednorozměrnými indexy a počet vláken bude stejný jako počet rovnic.

Pro dostatečné otestování bylo potřeba ještě vytvořit generátor rovnic, který je založen na náhodných datech. Aby vygenerovaná matice soustavy byla diagonálně dominantní, generují se dva typy dat. Pro pozice na diagonále a pro „pravé strany rovnice“ jsou generována větší čísla. Pro ostatní pozice je potřeba generovat malá čísla, a proto bylo rozmezí generovaných čísel zvoleno <0. 1).



Graf 4.1 Srovnání časů výpočtů rovnic pomocí Gauss-Seidelovy metody na CPU a GPU

4.1.4 Výsledky

Na grafu 4.1 můžeme porovnat časy výpočtu na klasickém procesoru a grafické kartě (v čase grafické karty je započtena režie kopírování z/do grafické paměti). Je vidět, že výkon grafické karty se projevuje až od 2048 rovnic (u 4096 rovnic můžeme zaznamenat 1,8-násobné zrychlení oproti CPU). Výpočet soustavy do tohoto počtu rovnic je rychlejší na CPU nebo se téměř shoduje s časem výpočtu na grafické kartě. Z grafu je také vidět, že CUDA (ale platí i obecně pro GPGPU) není vhodná pro

malý počet dat, protože výpočet s nízkým počtem dat (jednoduché operace ještě umocňují) je dokonce i pomalejší než na CPU. Ve výpočtu můžeme zaznamenat chybu v řádu miliontin, která je zapříčiněna odlišnou reprezentací čísel s plovoucí řádovou čárkou. Tato chyba se však s přibývajícím počtem rovnic téměř nemění.

4.2 Pohyb částic

Jednoduchá simulace pohybu (interakcí) částic v prostoru je založena na jejich aktuální pozici, rychlosti a její konstantní hmotnosti. Jejich pohyb není kromě vlastních srážek nijak omezený (tím je myšleno např. omezený prostor pohybu). Tento základní systém může být použit jako základ pro nejrůznější složitější částicové systémy. Může být využit pro pohyb v omezeném prostoru (toto může být využito v simulaci pohybu plynu v tlakové lahvi, meteorologii, metody konečných prvků, atd.), na který může působit určitá teplota, případně pohyb částic v elektromagnetickém poli atd.

4.2.1 Teoretický základ

Protože v našem případě se částice pohybují oproti rychlosti světla velmi pomalu, je využito klasické fyziky. K tomu abychom částice rozpohybovali, je potřeba částici umístěné v prostoru nastavit počáteční rychlost. Tímto mezi těmito částicemi vznikají síly, díky kterým mění jak svou pozici, tak i svou rychlost, případně i směr rychlosti (rychlost je vektorová veličina). Srážky se projevují na základě akce a reakce (neboli třetí Newtonův zákon), což znamená, že při srážce si částice navzájem vymění síly, kterými na sebe působí. Protože počítáme rychlosti a pozice v prostoru, tak musíme pro každý rozměr tyto parametry spočítat.

Vzdálenosti mezi jednotlivými částicemi spočítáme jednoduše tak, že provedeme rozdíl polohy dvou částic r_m a r_n, se kterými zrovna počítáme.

$$\begin{aligned} d_x &= x_m - x_n \\ d_y &= y_m - y_n \\ d_z &= z_m - z_n \end{aligned} \quad (4.13)$$

Pro výpočet síly také budeme potřebovat velikost vektoru vzdálenosti r. Ten vypočítáme jako druhou odmocninu ze součtu druhých mocnin jednotlivých vzdálenosti v prostoru.

$$\vec{r} = \sqrt{d_x^2 + d_y^2 + d_z^2} \quad (4.14)$$

Pokud tedy tento vypočítaný vektor není nulový, můžeme vypočítat celkovou sílu a zrychlení pro každý rozměr. Síla F se určí tak, že gravitační zrychlení g vynásobíme hmotností částice m a toto pak podělíme polohovým vektorem r.

$$F = \frac{m * g}{r} \quad (4.15)$$

Jednotlivá zrychlení a pak vypočteme pomocí této síly vynásobené vzdáleností d právě počítané dimenzi.

$$\begin{aligned} a_x &= F * d_x \\ a_y &= F * d_y \\ a_z &= F * d_z \end{aligned} \quad (4.16)$$

Tímto jsme si vypočítali důležité znalosti pro výpočet nové pozice a rychlosti částice. Nejdříve si vypočítáme rychlost *v*, abychom pak mohli určit i její pozici. K původní rychlosti tak připočteme rychlost vypočtenou z času *t*, po který se částice pohybuje, a její zrychlení *a*. Toto provedeme pro každý rozměr prostoru.

$$\begin{aligned} v_{x+1} &= v_x + t * a_x \\ v_{y+1} &= v_y + t * a_y \\ v_{z+1} &= v_z + t * a_z \end{aligned} \quad (4.17)$$

Nakonec vypočítáme nové pozice částice. Ty vypočítáme pomocí nově vypočítané rychlosti vynásobené časem a připočtené k předchozí pozici.

$$\begin{aligned} x_{n+1} &= x_n + t * v_{x+1} \\ y_{n+1} &= y_n + t * v_{y+1} \\ z_{n+1} &= z_n + t * v_{z+1} \end{aligned} \quad (4.18)$$

Tyto výpočty provedeme pro všechny částice našeho systému. V naší aplikaci můžeme zanedbat jednotky pod podmínkou, že nebudeme počítat s rychlostmi blížící se k rychlosti světla. To by již nemohla být použita klasická fyzika, ale relativistická a v tom případě by předcházející výpočty neplatily.

4.2.2 Implementace

Aplikace implementuje výše zmíněné výpočty na matici částic. Je zvolena matice kvůli tomu, že CUDA nepodporuje strukturu, která by se za normální situace dala použít a do kódu by vnesla i lepší čitelnost. Každý řádek obsahuje informaci o jedné částici, tj. postupně pozici a rychlost na x-ové, y-ové a z-ové souřadnici a hmotnost částice. Abychom mohli počítat se „starými“ hodnotami částic, je nezbytně nutné pro výpočet použít pomocnou matici částic, která si při každém kroku výpočtu vyměňuje adresu s maticí aktuálních pozic.

Algoritmus 4.3

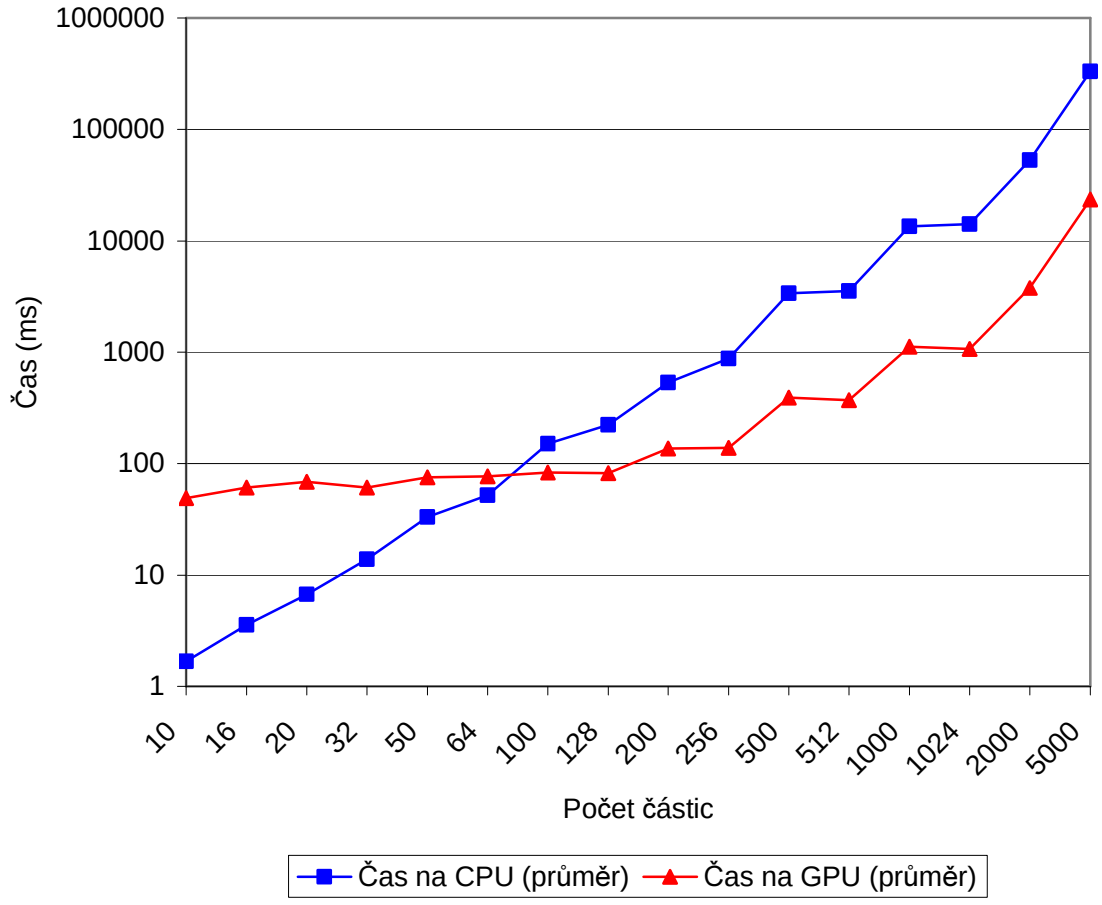
```
for (krok < počet kroků) {
  for (i < počet částic) {
    for (j < počet částic) {
      Vypočti vzdálenost mezi i-tou a j-tou částicí na ose x, y a z;
      Vypočti vektor vzdálenosti;
      if (polohový vektor ≠ 0) {
        Vypočti celkovou sílu působící na částici i od částice j;
        Vypočti zrychlení na ose x, y a z;
      }
    }
    Aktualizuj rychlost i-té částice na ose x, y a z;
    Aktualizuj pozici i-té částice na ose x, y a z;
  }
  Vyměň adresy matic;
}
```

Předchozí algoritmus je opět určen pro sériové zpracování. Paralelizaci provedeme tak, že každému vláknu přiřadíme jednu částici. Upravený kód znázorňuje následující algoritmus:

Algoritmus 4.4

```
for (krok < počet kroků) {
  for (i < počet částic) {
    Vypočti vzdálenost mezi i-tou a j-tou částicí na ose x, y a z;
    Vypočti polohový vektor;
    if (polohový vektor ≠ 0) {
      Vypočti celkovou sílu působící na částici i od částice j;
      Vypočti zrychlení na ose x, y a z;
    }
  }
  Aktualizuj rychlost částice vlákna n na ose x, y a z;
  Aktualizuj pozici částice vlákna n na ose x, y a z;
  Synchronizuj vlákna;
  Vyměň adresy částic;
  Synchronizuj vlákna;
}
```

Opět jako u výpočtu Gauss-Seidelovy rovnice nám odpadá použití jednoho for-cyklu.



Graf 4.2 Srovnání časů výpočtů pohybu částic na CPU a GPU

4.2.3 Výsledky

U této aplikace se mně vyskytly problémy s přesností. Na poslední chvíli jsem zjistil, že chyba není ve výpočtu, ale v tom, že data vypočtená na grafické kartě se nezkopírují na stranu procesoru. Bohužel tuto chybu se mně již nepodařilo opravit.

Výsledky se mně sice nepodařilo z grafické karty zkopírovat, ale máme k dispozici čas výpočtu. Zde se již citelně projevuje potenciál grafické karty. Testování probíhalo s tisíci částicemi a počet kroků (iterací), které aplikace provedla byla stanovena na 100 (tzn. kolikrát se částice posune ze své pozice). Celkový výpočet na GPU i s manipulací dat je 1131,8 ms. Zde je již vidět, že přenášení dat je opravdu časově velmi náročné. Zkusme nyní porovnat čas přenosu dat na grafickou kartu a přenos z ní. Čas strávený nahráváním dat na grafickou kartu činí 33,2 ms, zatímco nahrávání dat z grafické karty je 1034,4 ms. Zde se projevuje to, jak je grafická karta navržena. Při klasických grafických operacích je přenos dat na adaptér častější a proto je také více optimalizovaný než přenos opačný.

5 Závěr

V práci byly popsány knihovny pro obecné výpočty na grafických kartách. Objevily se zde především knihovny, které mají základ v jazyce C. Mohli jsme si všimnout, že mezi tyto knihovny patří i ty, které jsou vyprodukovány v současné době největšími výrobci grafických karet a to nVidia a AMD. Jejich knihovny však podporují pouze grafické karty ze stejné dílny.

Nejzajímavější knihovnou byla bezesporu knihovna StreamIt, která je produktem univerzity MIT. Její aplikace jsou tvořeny na základě fitlrů, které tvoří strukturu blokového schématu.

Knihovna CUDA, na kterou byla tato práce zaměřena, dokáže s kartami nVidia dosáhnout obrovského výkonu (nejvýkonnější adaptéry na trhu dosahují až 933 GFLOP). Navíc pokud se tyto karty spojí do režimu SLI, tak jejich výkon se teoreticky zdvojnásobí. Její komerční využití i její dosažené výsledky v těchto aplikacích značí, že CUDA je opravdu dobrým nástrojem pro práci s grafickými kartami. Největší zklamání přinesla oficiální dokumentace a referenční příručka přiložená v SDK. V těchto dokumentech jsou některé části nedostatečně popsány, případně úplně chybí.

Využitelnosti těchto knihoven do budoucna podle mého názoru není příliš jasná. Objevil se totiž nový standard, který chce sloučit techniku tohoto programování. Tento standard se nazývá OpenCL a je podporován ze strany nVidia i AMD. nVidia dne 20.4.2009 jako první zveřejnila SDK a ovladač pro podporu OpenCL

Demonstrační aplikace nebyly zaměřeny na optimalizaci, ale spíše na vyzkoušení funkčnosti a zrychlení oproti CPU. U druhé aplikace se bohužel vyskytly problémy s přenosem výsledků z grafické paměti. Zrychlení však bylo znatelné. U simulace pohybu částic se oproti CPU výpočet provedl až 13x rychleji (do času výpočtu na straně grafické karty je započtena i manipulace s daty, která má výrazný vliv na celkový čas).

V této práci jsem byl omezen pomalým hardwarem. Při použití grafické karty nVidia GeForce 9400 GT (44,8 GFLOP) nelze čekat nějaké zázraky v rychlostech. Tento adaptér také nepodporuje výpočty s čísly s plovoucí řádovou čárkou s dvojitou přesností.

Z pohledu dalšího vývoje je možné navázat na aplikaci, která simuluje pohyb částic. Jejím optimalizováním a rozšířením například o chování plynných částic v uzavřeném prostoru za různých teplot a rozměrech prostoru by pak vznikla zajímavá aplikace. Jistě by také bylo velmi zajímavé vytvořit aplikaci na kódování dat.

Literatura

[1] Wikipedia: GPGPU - Wikipedia, The Free Encyclopedia. 2009. [navštíveno 21.4.2009]. URL: <http://en.wikipedia.org/wiki/GPGPU>.

[2] CHIP: Magazín informačních technologií. č. 6 (červen 2007). Praha: Vogel, 2007. Vychází měsíčně. ISSN 1210-0684.

[3] Internetové stránky GPGPU. [navštíveno 21.4.2009]. URL: <www.gpgpu.org>.

[4] Matt Pharr, Randima Fernando: GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley. 2005. ISBN 0-321-33559-7.

[5] João Luiz Dihl Comba, Carlos A. Dietrich, Christian A. Pagot and Carlos E. Scheiddegger: Computation on GPUs: From a Programmable Pipeline to an Efficient Stream Processor. 2003. < http://www.inf.ufrgs.br/~comba/papers/2003/gpu.pdf>.

[6] Technical Brief: nVidia GeForce GTX 200 GPU, Architectural Overview. <http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf>

[7] NVIDIA Tesla: GPU Computing Solutions for HPC. <http://www.nvidia.com/docs/IO/43395/tesla_product_overview.pdf>.

[8] nVidia Corporation: NVIDIA CUDA Programming Guide 2.2. <http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf>.

[9] Randima Fernando, Mark J. Kilgard: The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics. Addison-Wesley. 2003. ISBN 0-321-19496-9.

[10] RNDr. Břetislav Fajmon, Ph.D., Mgr. Irena Růžičková: Matematika 3.

[11] Wikipedia: Cloce to Metal. [navštíveno 10.5.2009]. <http://en.wikipedia.org/wiki/Close_to_Metal>.

[12] Internetové stránky RapidMind. [navštíveno 10.5.2009]. <http://www.rapidmind.net>.

[13] Internetové stránky EcoLib. [navštíveno 11.5.2009]. <http://www.gpucomputing.eu>.

[14] Internetové stránky StreamIt. [navštíveno 11.5.2009]. <http://www.cag.csail.mit.edu/streamit/>.

[15] Matthew Monteyne: The RapidMind Multi-core Development Platform. 2009. <http://www.rapidmind.net/pdfs/WP_RapidMindPlatform.pdf>.

[16] Internetové stránky OpenCL. [navštíveno 11.5.2009]. <http://www.khronos.org/opencl/>.

[17] Intel: Intel® microprocessor export compliance metrics. <http://www.intel.com/support/processors/sb/CS-023143.htm>.

A. Výpočetní způsobilost grafické karty

Výpočetní způsobilost (compute capability) [8] je vyjádření obecné specifikace a vlastností grafické karty, která je zapotřebí pro výpočty na architektuře CUDA. Každá vyšší způsobilost dědí vlastnosti nižší způsobilosti.

Specifikace pro výpočetní způsobilost 1.0:

- Maximální počet vláken na blok je 512
- Maximální velikost x-ové, y-ové a z-ové souřadnice jednoho vlákna je 512 x 512 x 64
- Maximální velikost každého rozměru mřížky (gridu) je 65535
- Velikost warpu* je 32 vláken
- Počet registrů na multiprocesor je 8192
- Velikost sdílené paměti dostupné pro multiprocesor je 16 kB. Ta je organizovaná do 16 bank.
- Celková velikost konstantní paměti je 64 kB
- Vyrovnávací paměť pro konstantní paměť je 8 kB na multiprocesor
- Vyrovnávací paměť pro paměť na textury se pohybuje mezi 6 a 8 kB na multiprocesor
- Maximální počet aktivních bloků na multiprocesor je 8
- Maximální počet aktivních warpů na multiprocesor je 24
- Maximální počet aktivních vláken na multiprocesor je 768
- Pro jednorozměrné textury vázané na CUDA pole je maximální šířka 2^{13}
- Pro jednorozměrné textury vázané na lineární paměť je maximální šířka 2^{27}
- Pro dvourozměrné textury vázané na CUDA pole nebo na lineární paměť je maximální šířka 2^{16} a maximální výška 2^{15}
- Pro třírozměrné textury vázané na CUDA pole je maximální šířka 2^{11}, maximální výška 2^{11} a maximální hloubka 2^{11}
- Limit na velikost jádra činí 2 miliony instrukcí
- Každý multiprocesor je složen z 8 procesorů, z toho vyplívá, že multiprocesor je schopen vykonat 32 vláken warpu za 4 hodinové cykly

Specifikace pro výpočetní způsobilost 1.1:

- Podpora atomických funkcí pracující s 32-bitovými slovy v globální paměti

Specifikace pro výpočetní způsobilost 1.2:

- Podpora atomických funkcí pracující s 64-bitovými slovy v globální paměti
- Počet registrů na multiprocesor je 16384
- Maximální počet aktivních warpů na multiprocesor je 32
- Maximální počet aktivních vláken na multiprocesoru je 1024

Specifikace pro výpočetní způsobilost 1.3:

- Podpora čísel s dvojitou přesností v plovoucí řádové čárce

* warp je skupina 32 vláken. Postupně se střídají všechny warpy, nad celým warpem se provede jedna instrukce a pokračuje další warp.

B. Testovací stroj

Testování aplikací probíhalo na stoji s následujícími parametry (jsou zmíněny pouze důležité parametry):

- Procesor Intel Core 2 Duo E8400 3Ghz (24 GFLOPS)
- 2048 MB DDR2 800MHz CL5
- nVidia GeForce 9400GT 512 MB GDDR2 s výpočetní způsobilostí 1.1

Podrobnější informace o grafické kartě (pouze ty, co nejsou vyjádřeny výpočetní způsobilostí):

- Frekvence hodin – 1,4 GHz
- Teoretický výkon – 1,4 (frekvence hodin) x 2 (počet multiprocesorů) x 32 (počet vláken ve warpu) / 2 (počet cyklu na instrukci nad warpem) = 44,8 GFLOPS
- 128 bitů šířka rozhraní
- Šířka pásma – 12,8 GB/s

C. Sčítání matic s využitím CUDA

Tento příklad je zkrácenou verzí. Celá a funkční verze je na přiloženém CD.

```
// Kernel pro sečtení matice A s maticí B. Její výsledek je uložen
// v matici C
__global__
void addMatrixKernel(float *A, float *B, float *C, unsigned int width){
      // index vlákna
      unsigned int tx = threadIdx.x;
      unsigned int ty = threadIdx.y;

      // index bloku
      unsigned int by = blockIdx.y;

      // výpočet jednotlivých submatic pro bloky vlákna
      unsigned int begin = width * BLOCK_SIZE * by;
      unsigned int end = begin + width - 1;
      unsigned int step = BLOCK_SIZE;

      // součet matic
      for (unsigned int i = begin; i < end; i += step)
            C[i+width*ty+tx] = A[i+width*ty+tx] + B[i+width*ty+tx];
}

int main(int argc, char** argv) {
    // velikost matice
    long matrix_size = MATRIX_WIDTH * MATRIX_HEIGHT;
    // velikost matici v paměti
    long mem_matrix_size = sizeof(float) * matrix_size;

    // alokace paměti pro matice na straně CPU
    float *h_A = (float *) malloc(mem_matrix_size);
    float *h_B = (float *) malloc(mem_matrix_size);

    // naplnění matic daty
    initiateMatrix(h_A, matrix_size);
    initiateMatrix(h_B, matrix_size);

    // alokování paměti na straně GPU
    float *d_A;
    cutilSafeCall(cudaMalloc((void **) &d_A, mem_matrix_size));
    float *d_B;
    cutilSafeCall(cudaMalloc((void **) &d_B, mem_matrix_size));

    // alokování paměti pro výslednou matici na straně GPU
    float *d_C;
    cutilSafeCall(cudaMalloc((void **) &d_C, mem_matrix_size));

    // zkopírovaní dat z hlavní paměti na grafickou kartu
    cutilSafeCall(cudaMemcpy(d_A, h_A, mem_matrix_size,
cudaMemcpyHostToDevice));
```

```
    cutilSafeCall(cudaMemcpy(d_B, h_B, mem_matrix_size,
cudaMemcpyHostToDevice));

    // nastavení mřížky a zavolání kernelu
    dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
    dim3 grid(MATRIX_WIDTH / threads.x, MATRIX_HEIGHT / threads.y);
    addMatrixKernel<<<grid, threads>>>(d_A, d_B, d_C, MATRIX_WIDTH);

    // alokování paměti pro výslednou matici na straně CPU
    float *h_C = (float*) malloc(mem_matrix_size);

    // zkopírování výsledků z grafické paměti do hlavní paměti
    cutilSafeCall(cudaMemcpy(h_C, d_C, mem_matrix_size,
cudaMemcpyDeviceToHost));

    // uvolnění paměti
    free(h_A);
    free(h_B);
    free(h_C);
    cutilSafeCall(cudaFree(d_A));
    cutilSafeCall(cudaFree(d_B));
    cutilSafeCall(cudaFree(d_C));
}
```

D. Obsah přiloženého média

- Text této práce
- Zdrojové soubory
- Instalační soubory knihoven
- Příručka pro instalaci knihoven a implementovaných příkladů
- Grafy, použité v této zprávě