ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická

DIPLOMOVÁ PRÁCE

2009 Jindřich GOTTWALD

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Jindřich Gottwald**

Magisterský studijní program: Elektrotechnika a informatika, dobíhající navazující

Obor: Výpočetní technika

Název: **Matematické a fyzikální výpočty na grafických kartách**

Pokyny pro vypracování:

V praxi otestujte vhodnost a použitelnost běžně využívaných grafických karet k numerickým výpočtům z oblasti matematické algebry. Zaměřte se na platformu MS Windows, porovnejte přístup pomocí CPU / DirectX 9 / DirectX 10. Soustřeďte se na přímé programování shaderů grafické karty (např. pomocí hlsl) a neberte v potaz jazyky specializované na numerické výpočty (Brook, Cg). Práci doplňte analýzou možností využití grafických karet pro fyzikální výpočty v moderních počítačových hrách (včetně specializovaného HW: Ageia PhysX, nVidia Tesla, ATI FireStream).

Seznam odborné literatury: Dodá vedoucí práce

Vedoucí: Ing. Ivan Šimeček

Platnost zadání do: letní semestr 2008/09

Prof. Ing. Pavel Tvrdík, CSc.
vedoucí katedry počítačů

Doc. Ing. Boris Šimák, CSc.
děkan

V Praze dne: 30. 1. 2008

České vysoké učení v Praze
Fakulta elektrotechnická

Diplomová práce

Matematické a fyzikální výpočty na grafických kartách

Jindřich Gottwald

Vedoucí práce: ing. Ivan Šimeček

Studijní program: Elektrotechnika a informatika dobíhající magisterský
Obor: Informatika a výpočetní technika
květen 2009

Poděkování

Tato práce zabrala nemalé množství času nejen mně, ale i několika dalším lidem, za což jim patří můj dík (v abecedním pořadí):

Jan Hloušek, Jakub Hloušek, Petr Matěja, Pavel Matys, Jan Ondřich, Martin Ouvín, Vladislav Spevák, Ivan Šimeček, Martin Živný.

Maximálně podporující rodinu a přátele snad ani jmenovat není potřeba (oni vědí).

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci na téma "Numerické výpočty na grafických kartách" vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.

Nemám závažný důvod proti nekomerčnímu užití tohoto školního díla ve smyslu §60 Zákona č.121/2000 Sb. ,o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze 1. dubna 2008 ……..……..

Jindřich Gottwald

Abstrakt

Cílem této diplomové práce je praktické využití běžně používaných grafických karet současnosti v oblasti numerických výpočtů. Projekt je zaměřen na platformu MS Windows s využitím DirectX a jejich vyššího jazyka HLSL, určeného k programování shaderů. Nastiňuje také možnost simulování fyziky na grafických kartách v moderních počítačových hrách.

Abstract

The goal of this thesis is to show and test usability of nowadays graphical cards for numeric math computation. The project is based on MS Windows platform using DirectX and its High Level Shader Language (HLSL). It also outlines the possibility of using graphic cards for the simulation of physics in modern computer games.

Obsah

Obrázky

Tabulky a grafy

1. Úvod

Výpočetní kapacita moderních grafických karet láká spousty vývojářů k nestandardnímu využití (počítání fyziky, umělé inteligence, simulace kapalin a k obecným numerickým výpočtům). Na soudobém hardwaru a softwaru je toto programování možné, trpí ovšem několika neduhy a omezeními, které vyplývají z původního technologického zaměření a bude zmíněno níže. Hlavní výrobci grafických karet udávající trendy (nVidia, ATI) nezůstávají ovšem k trhu slepí a usilovně vyvíjejí přístupnější hardware, určený nejen ke grafickým výpočtům. Pro tuto problematiku se vžila zkratka *GPGPU* (*general purpose computation on graphics processing units* neboli *obecné výpočty na grafických čipech*).

Stejně tak nové DirectX 11 slibují silnější a příjemnější rozhraní. V této práci ovšem nebudeme řešit blízkou budoucnost, ale zaměříme se na nejpoužívanější software a hardware současnosti, využijeme tudíž DirectX9 a Shader model 2.0. Přístup DX9 k GPGPU porovnáme s nastupující generaci DirectX10 s Shader modelem 4.0. S úspěchem také uplatníme vyšší jazyk shaderů HLSL, který se v DirectX objevuje právě od verze 9.

První kapitolu tvoří tento úvod. Druhá lehce poodkryje hardwarovou základnu pro GPGPU. Ve třetí kapitole probereme teoretické základy a následně jejich implementaci v DirectX. V případě nejasností doporučuji prostudovat MSDN uvedených příkazů a podívat se na jejich použití v praxi v další, tedy čtvrté kapitole. V té je rozebrán klasický algoritmus Gaussovy eliminace pro CPU v porovnáním s řešením pod DirectX. Jádrem této práce je kapitola pátá, ve které dojde na zhodnocení naměřených výsledků. Šestá kapitola je nahlédnutí do možné budoucnosti a v sedmé se v krátkosti podíváme pod pokličku fyzikálních *enginů* moderních her.

Osmá kapitola je povinný závěr a v deváté uvádím přehled internetových odkazů užitečných při psaní tohoto veledíla. Stojí za to zmínit, že v elektronické podobě této práce jsou všechny odkazy na použitou literaturu klikatelné a vedou přímo do použitého článku. V dodatkové desáté kapitole dojde kromě výčtu použitého softwaru a popisu obsahu CD, také k vysvětlení použitých zkratek a shrnutí nejasných termínů do minislovníčku.

2. Hardwarová stránka věci

2.1. jak to všechno začalo

Vynecháme úplné počátky, kdy měly grafické karty dost práce s 2D grafikou a začneme popisem let 90tých, kdy se začaly objevovat 3D akcelerátory hardwarově podobné tomu, co používáme dodnes.

Technologii tlačily kupředu zejména počítačové hry, u zrodu 3D akce stály: *Ultima Underworld (1992 Blue Sky Production), Wolfenstein 3D (1992 idSoftware), DOOM (1993 idSoftware), System Shock (1994 Looking Glass Technologies), The Terminator: Future Shock (1995 Bethesda)*. Úspěch zmíněných titulů ukázal výrobcům hardwaru, za co budou hráči ochotni utrácet nemalé množství peněz. V roce 1996 vydává společnost *idSoftware* další revoluční hru *Quake*. Byla to první hra kompletně ve 3D s přímou podporou hardwarové akcelerace (*OpenGL* emulované pod *3Dfx Glide*).

Profesionální využití, jako jsou CAD systémy nebo filmová studia by samy takového boomu v oblasti 3D akcelerace neměly šanci dosáhnout.

Pro širší veřejnost začala historii 3D grafických akcelerátorů psát firma 3Dfx se svým čipem Voodoo, který vyšel v roce 1996. Nejednalo se sice o první 3D čip, ale jeho masové rozšíření bylo do té doby nevídané a potvrdilo, jak velký zájem o 3D akcelerátory uživatelé (rozuměj – hráči počítačových her) mají. Kromě zmíněné *3Dfx* se v té době snažily vlastní 3D akcelerační čip dodat na mainstream trh především tyto firmy: *ATI, Chromatic, Matrox, nVidia, Oak, Rendition, S3*. Časem některé zkrachovaly, byly odkoupeny, nebo zaměřily vývoj jiným směrem.

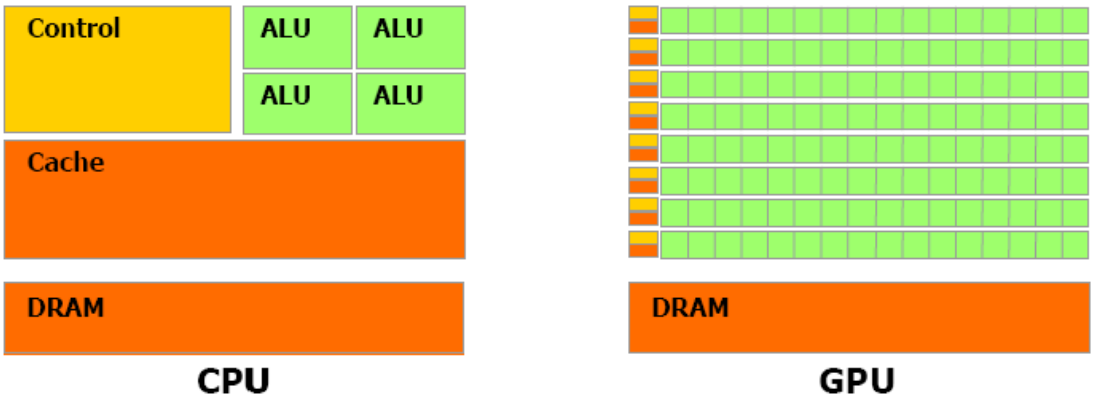
Pokud se dnes (2009) podíváme na tento seznam vidíme, že z tvrdého konkurenčního boje vzešli dva vítězové. Jde o ATI/AMD a nVidia, dva soupeřící giganty. Samozřejmě, že mnoho odborníků ze zaniklých firem pracuje nyní právě pro tyto dvě společnosti.

Zájemce o vyčerpávající popis historie grafických karet odkáži na výtečné články Jiřího Součka zveřejněné na [24]. Jak vypadá hardware vlajkových lodí v podání ATI a nVidia uvidíte v poslední podkapitole. Historie DirectX a shader modelů je popsána v kapitole zabývající se softwarem.

2.2. CPU vs GPU

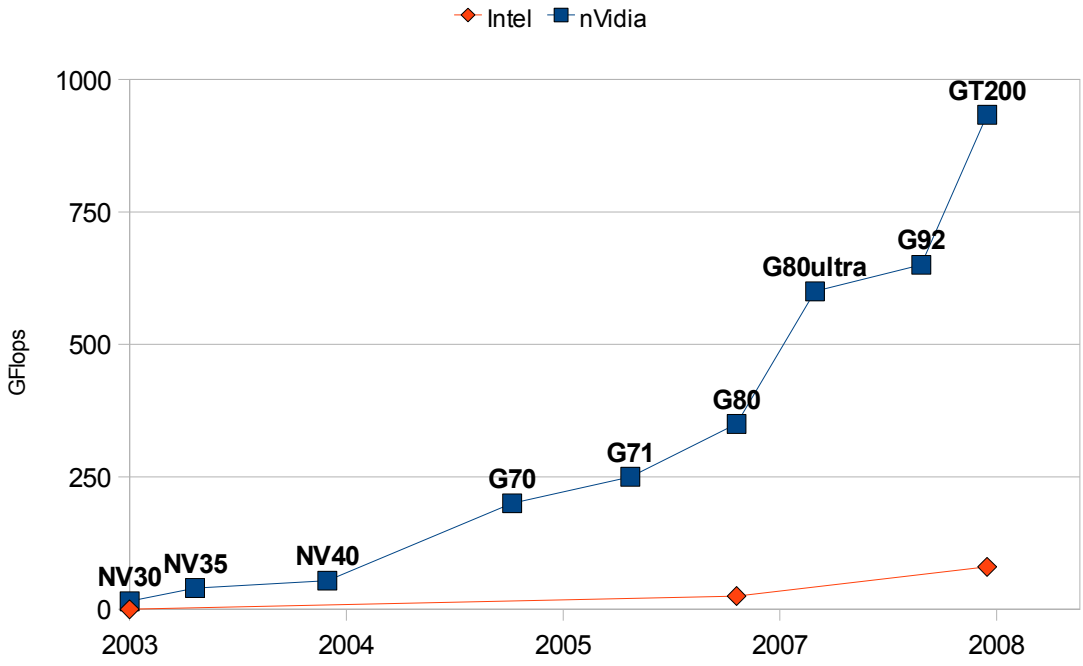
Velice jednoduše řečeno, jde při zpracování 3D scény převážně o výpočet jednoduchých aritmetických operací a práci s vektory a maticemi. Operace se provádějí nad obrovským množstvím dat. Tato jednoúčelovost je znát v návrhu grafických karet, které tedy obsahují velké množství relativně jednoduchých aritmetických jednotek, optimalizovaných pro tyto specifické operace.

Naproti tomu CPU se musí vypořádat s obecnými problémy, jeho ALU jsou tedy daleko komplexnější. Zjednodušené porovnání zmíněných přístupů je znázorněno na obr.1.

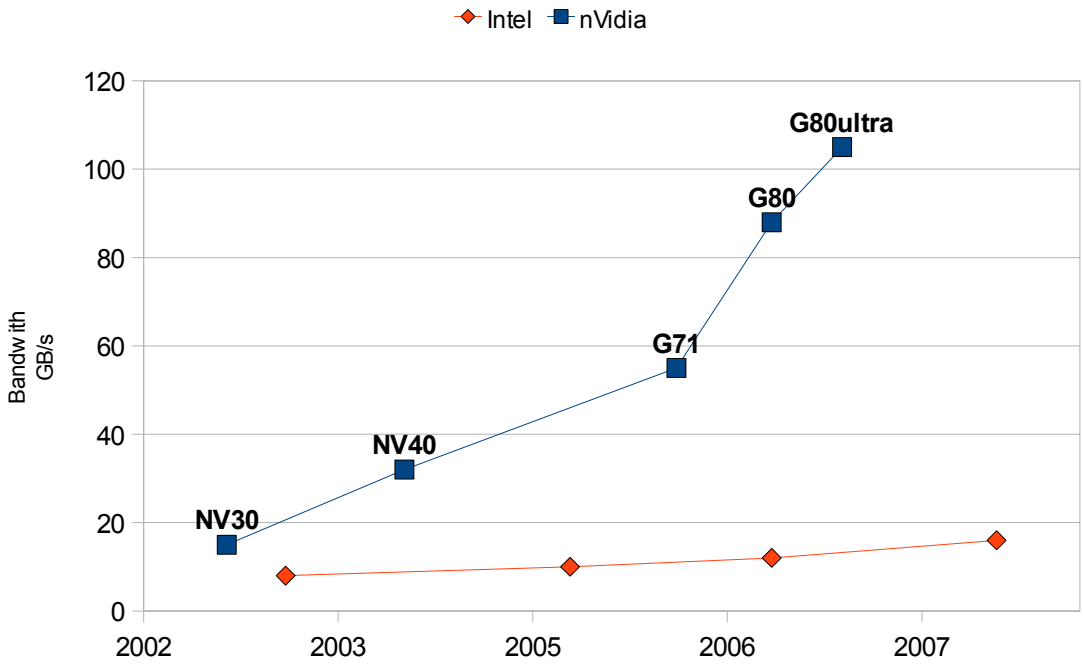


Obr 1: architektura CPU vs GPU (převzato z [33])

V dnešní době dochází k rozmachu GPGPU výpočtů, výrobci grafických karet se snaží přidávat podporu obecnějších výpočtů a zároveň při tom ponechat výhody původního konceptu. Stejně tak CPU prošlo spoustou optimalizací pro zvýšení paralelizmu, za zmínku stojí procesor *Cell* [10]. Jak je vidět na obr.2 a obr.3, bude-li se výkon zvyšovat stejným tempem jako doposud, máme se na co těšit. Na zmíněných obrázcích jsou uvedeny pouze společnosti Intel a nVidia, jde jen o zjednodušení grafu, konkurence v podání AMD/ATI samozřejmě roste stejným tempem.



Obr 2: CPU vs GPU nárůst výkonu



Obr 3: CPU vs GPU nárůst propustnosti

2.3. co se děje při klasickém renderování scény

Moderní grafická karta se skládá z několika částí, jde o: systémové rozhraní, paměť, procesor (GPU), frame buffer, výstupní převodník.

Data (textury, objekty) vstupují do grafické karty skrz její rozhraní (PCI, AGP, PCIe), kde jsou ukládána do grafické paměti. Z nich se v srdci grafické karty (GPU) vypočítá pozice objektů ve 3D scéně, namapují se textury, celá scéna se převede do 2D a výsledek se uloží do frame bufferu. Z něj výsledný obraz putuje skrze převodníky směrem k obrazovce. Mezi nejpoužívanější výstupní převodníky patří RAMDAC (analogový výstup pro CRT či LCD), TMDS (digitální LCD, HDMI) nebo TV-out encoder.

Nejzajímavější práci odvádí bezpochyby GPU. Ke zpracování dat dochází postupně v jednotlivých částech grafické *pipeline.* Typický průchod dat skrz pipeline je znázorněn na obr.4. K jeho pochopení je potřeba si uvědomit, že každá trojrozměrná scéna se skládá z množství trojúhelníků, udávajících tvar a pozici objektů.

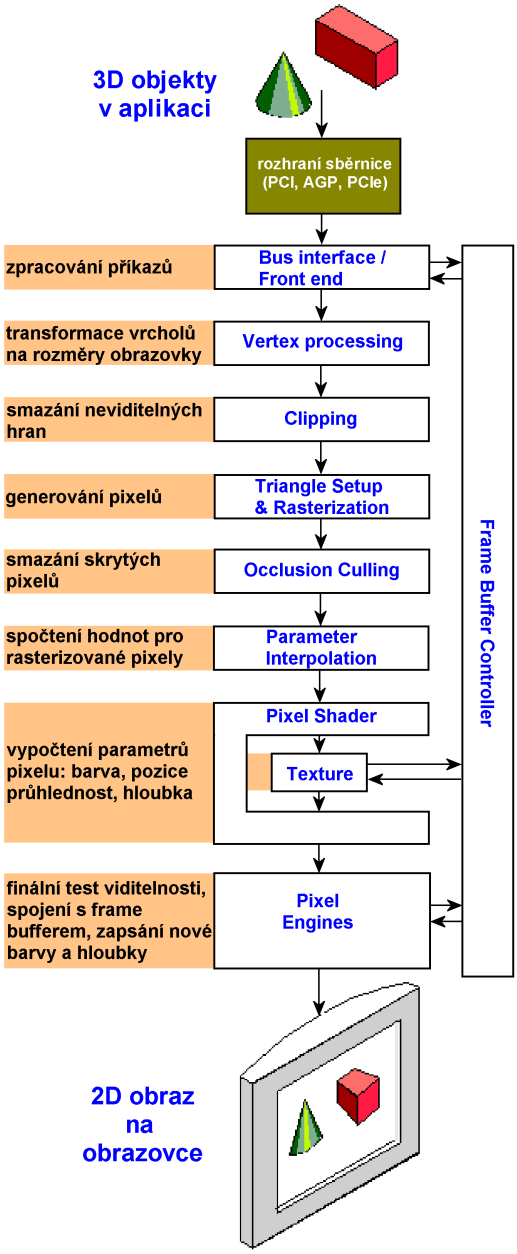
Obr 4: pipeline obecně (upraveno z [21])

1. Vertex shader vytvoří ze souřadnic a vektorů 3D scénu.
2. Neviditelné prvky jsou odstraněny ze scény.
3. Scéna je nasvícena jedním nebo více světelnými zdroji.
4. Pixel shader a texturovací jednotky obarví jednotlivé objekty scény.
5. Anti-aliasingem jsou vyrovnány hrubé a ostré hrany objektů.

Konkrétní vnitřní reprezentace pipeline se liší podle výrobce a použité generace grafické karty a bude zmíněna v dalších podkapitolách.

2.4. GPGPU pod DirectX9

Pro naše výpočty musíme použít stejnou pipeline, jako v případě 3D aplikací. Možnosti některých jejích bloků ovšem vůbec nevyužijeme a naopak se budeme snažit softwarově minimalizovat jejich vliv. Obrázek 5 znázorňuje a popisuje pipeline DirectX9.



Obr 5: pipeline DX9 (upraveno z [9])

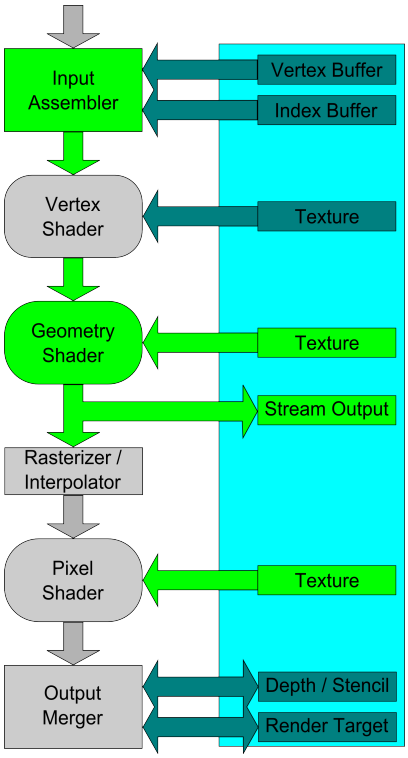
Pro GPGPU jsou zajímavé dvě části a to *Vertex processing* a *Pixel shader*.

Vertex procesor se standardně stará o konverzi vstupních vrcholů na 2D plochu. Vrcholy díky většímu počtu jednotek zpracovává paralelně. Jedná se o programovatelný procesor, v této diplomové práci ho ovšem nebude zapotřebí. Vše totiž obstará následující procesor, a tím je:

Pixel shader, který standardně mapuje textury na fragmenty posílané z rasterizační jednotky, proto se mu také někdy říká *fragment shader*. Samozřejmě také paralelně. Jeho programovatelnost nám umožňuje pracovat s atributy pixelů, jako je např. barva. Pixel procesory bývají výpočetně silnější (mají víc paralelních jednotek) oproti vertexovým.

2.5. změny DirectX10 oproti DirectX9

K hlavním novinkám patří přidání dalšího typu shader procesoru zvaného *geometry shader*. Jak je vidět z obr.6, řadí se v pipeline ihned za vertex shader. Umožňuje generovat grafická primitiva (body, přímky, trojúhelníky). Pro GPGPU pro něj bohužel nenajdeme využití.

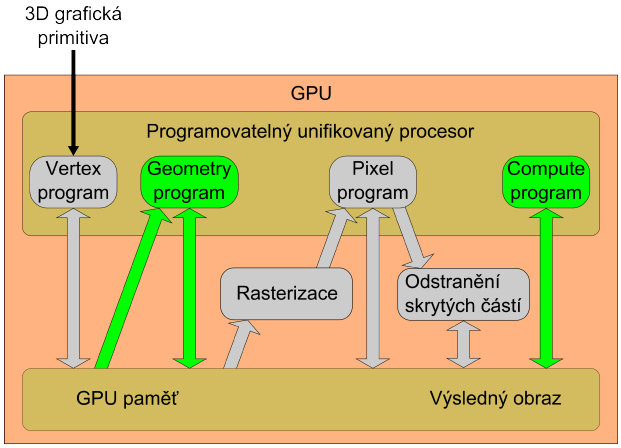


Obr 6: pipeline DX10

(upraveno z [25])

Naopak poměrně velký význam pro GPGPU aplikaci má požadavek na přesnost výpočtů v plovoucí čárce. Minimální definovaná přesnost v DX10 je FP32 oproti FP16 v DX9 a podle [46] se práce s plovoucí čárkou přiblížila standardu IEEE 754. O přesnosti více pojednává kapitola 2.7.

Poslední markantní změnou je spojení všech shaderů do jednoho, takzvaného *unifikovaného shaderu*. Toto spojení nemusí být nutně na hardwarové úrovni, nicméně většina karet podporujících DX10 má shader jednotky skutečně hardwarově unifikovány do jediné. Tím dojde k lepšímu využití všech shader procesorů což by i v GPGPU praxi mělo přinést zrychlení. Ze softwarové stránky znamená unifikovaný shader jednotnou instrukční sadu pro všechny typy shaderu. Jak je vidět na obrázku 7, tok dat pipelinou zůstal nezměněn.



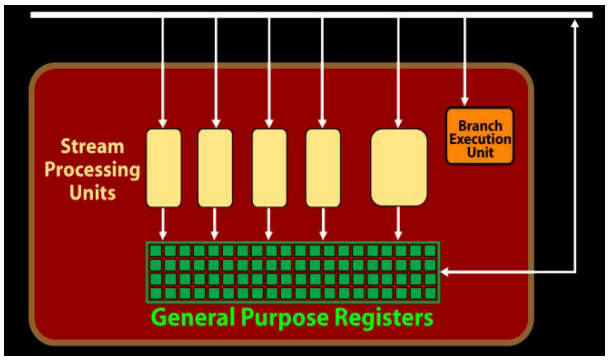
Obr 7: unified shader (upraveno z [26])

2.6. rozdíl přístupu ATI a nVidia

Nabízí se porovnání posledního (rok 2009/Q1) vydaného hardwaru od obou firem. ATI staví na svém čipu RV770, jehož technologický ekvivalent je GT200 od nVidia. Oba čipy byly v rámci této práce testovány, výsledky jsou zaznamenány v kapitole 5.2. Jde o karty ATI Radeon HD 4870 nVidia GeForce GTX280.

a) ATI

Dá se říci, že ATI má momentálně technologický náskok a o RV770 se ve většině recenzí (např. [19]) mluví v superlativech. Stejně jako u předchozího čipu R600 se jedná o superskalární architekturu, kdy se každý *shader procesor* (SP) skládá z pětice ALU neboli *stream procesorů.*



Obr 8: RV770 5D ALU (převzato z [19])

Na obr.8 je jedna z pěti jednotek SP znázorněna ve větší velikosti. Není to náhoda, této ALU jednotce se říká *special function unit* (SFU) a stará se mimo jiné o speciální funkce, jako je sin, cos, log apod. K tomu zvládá většinu aritmetických funkcí, co ony menší SP. Každá ALU v jednoduché přesnosti dokáže spočítat dvě instrukce sčítání nebo násobení. V případě výpočtů ve dvojité přesnosti (FP64) se čtveřice ALU rozdělí na dvě dvojice a každá dvojice spočítá násobení/sčítání v jednom taktu. SFU může být v tom samém taktu využita na FP32 výpočet. Celý čip RV770 obsahuje 160 SP, což znamená celkem 800 ALU běhajících na 750MHz. Pro jednoduchou přesnost dostáváme teoretickou rychlost:

*750 MHz/s * 160 SP * 5 ALU * 2 flop/cycle =* ***1,2 TFlops***

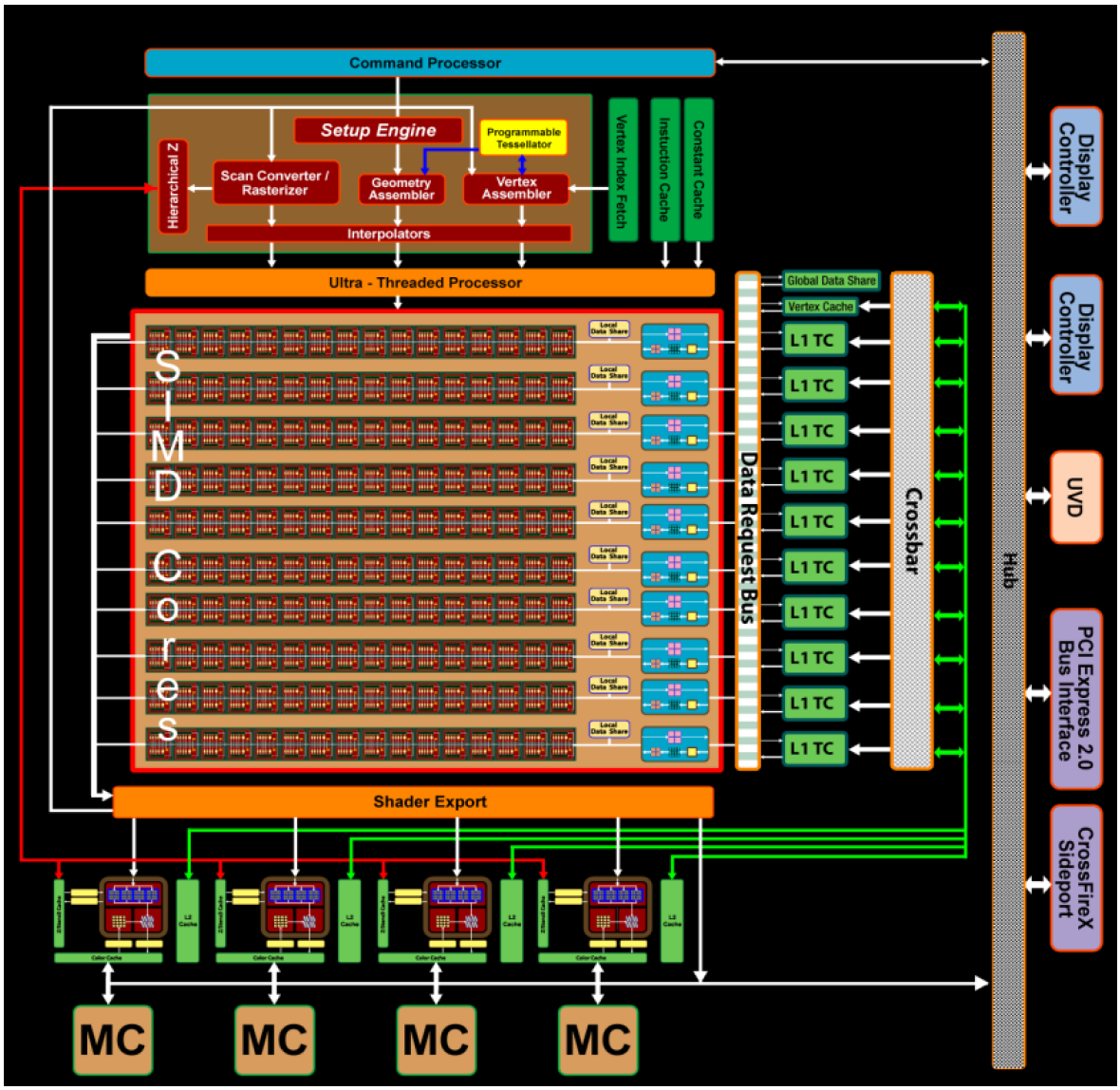
Použijeme-li dvojitou přesnost, dostaneme:

*750 MHz/s * 160 SP * 2 ALU_FP64 * 1 flop/cycle =* ***240 GFlops***

Zároveň s tím můžeme využívat SFU v jednoduché přesnosti rychlostí:

*750 MHz/s * 160 SP * 1 ALU_SFU * 2 flop/cycle =* ***240 GFlops***

Struktura čipu je na obr.9, kde napočítáme 10 SIMD jader. Jedno SIMD jádro dokáže vykonat stejnou aritmetickou operaci zároveň nad větším množství dat, nedokáže však provádět odlišné úkoly. Každé jádro se skládá právě z 16ti SP jednotek a je propojeno s jednou *texturovací jednotkou* (TU – na obr.9 modře). Zvyšování výkonu shaderů na stávající architektuře tedy znamená přidání kombinace SIMD jádra s TU.

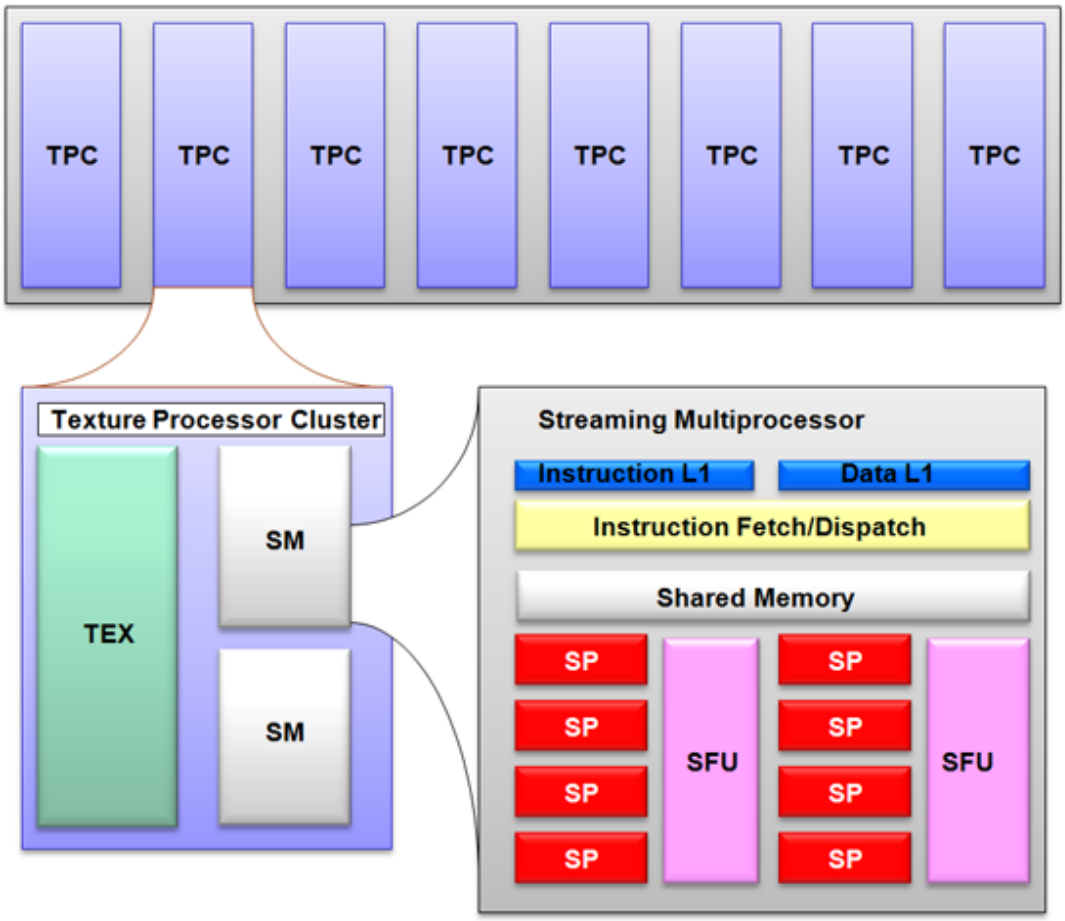


Obr 9: RV770 architektura (převzato z [28])

Detailnější popis architektury je nad rámec této práce, snad jen zmíním, že celý čip obsahuje 956 milionů tranzistorů. Více o R600 naleznete v [20], zajímavé články a recenze RV770 pak v [19], [28], [29] a [30].

b) nVidia

Předchůdcem čipu GT200 byl opěvovaný G80, nVidia pokračovala v evoluci ozkoušeným způsobem. Ten je založen na škálovatelném procesorovém poli (*scalable processor array* = SPA), které se dále dělí, viz obr.10. Největší jednotkou jsou *texture processing cluster* (TPC). O rozdělení výpočtů mezi jednotlivé TPC se stará hardwarový *thread scheduler* (TS) neboli plánovač výpočetních vláken. Každý TPC obsahuje několik *streaming multiprocesorů* (SM) a paměť. SM je složen z *stream procesorů* (SP), *special function unit* (SFU) a sdílené lokální paměti. Stream procesor bývá také označován jako *thread procesor* nebo ALU. Tento systém založený na SPA umožňuje výbornou škálovatelnost a další generace GPU od nVidia na něm bude pravděpodobně také založena. Konkrétně GT200 obsahuje 10 TPC, každá se 3 SM. A konečně SM má 8 SP a 2 SFU, celkem tedy 240 stream procesorů a 60 SFU. Celý čip má 1,4 miliard tranzistorů.



Obr 10: G80 (GT200) architektura (převzato z [31])

Co na obrázku není znázorněno a čím se nVidia příliš nechlubí je jednotka pro výpočet ve dvojité přesnosti, každý SM obsahuje jednu a při jejím použití je všech osm SP vypnuto. Stejně jako klasický stream procesor dokáže i tato jednotka spočítat dvě instrukce za takt. SFU kromě standardních operací (viz podkapitola o ATI) zvládne za takt provést jednu operaci se čtyřsložkovým vektorem.

Z výše zmíněných informací je možné vypočítat teoretické rychlosti GT200. Grafické karty založené na tomto čipu se liší v počtu vypnutých TPC jednotek a především v taktovací frekvenci. Pro náš výpočet vyjdeme z GTX280, která má všechny TPC odemčeny a standardně běží na 1296MHz. Pro jednoduchou přesnost dostaneme:

*1296 MHz/s * (240 SP * 2 flop/cycle + 60 SFU * 4 FPU/SFU * 1 flop/cycle) =* ***933 GFlops***

Při počtech ve dvojité přesnosti spadne výkon na:

*1296 MHz/s * 30 SM * 2 flop/cycle =* ***78 GFlops***

Při využití SP pro výpočty ve dvojité přesnosti se zůstávají nevyužité SFU, na nich je možné zároveň počítat v jednoduché přesnosti rychlostí:

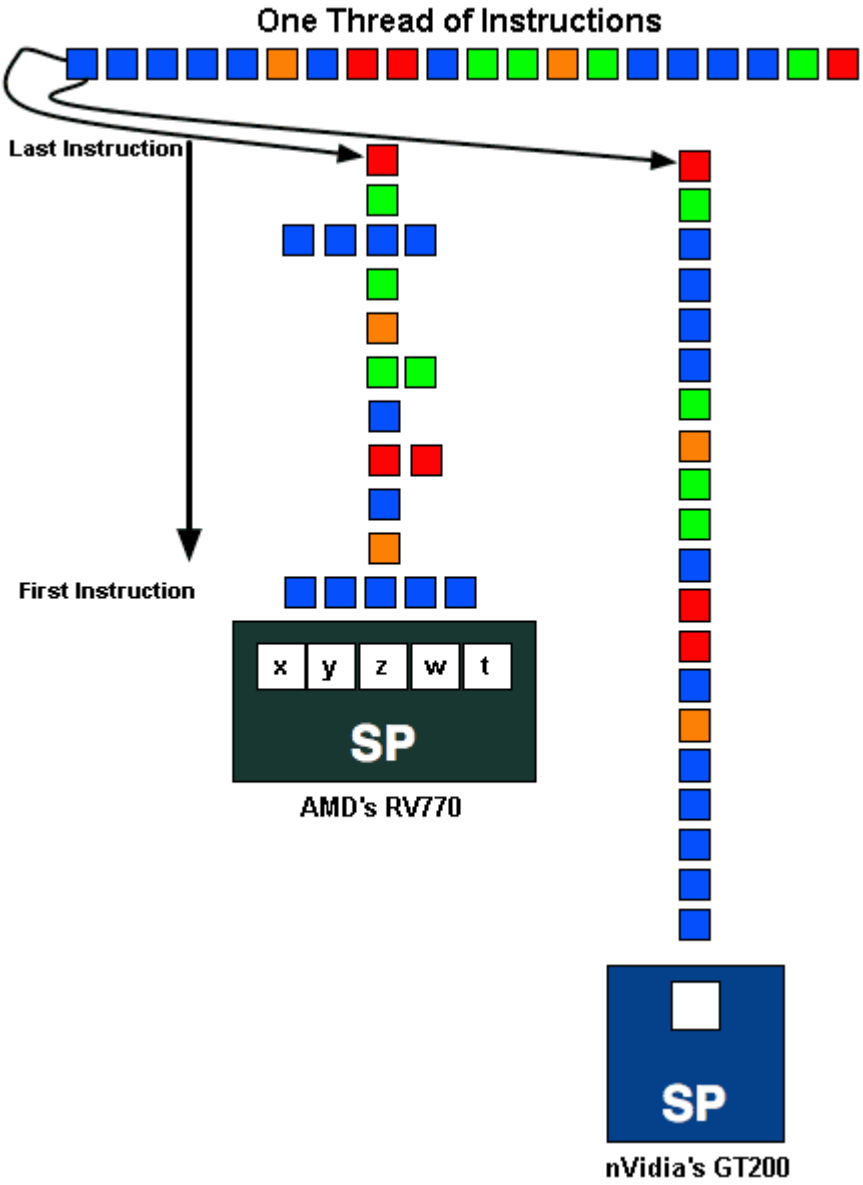
*1296 MHz/s * 60 SFU * 4 FPU/SFU * 1 flop/cycle =* ***311 GFlops***

Každý shader multiprocesor má jednu *instruction unit* (IU) která dokáže řídit 32 paralelních vláken. Tuto skupinu vláken nVidia nazývá *warp* a jejich počet odpovídá celkovému počtu SP uvnitř SM. Způsob zpracování instrukcí nVidia nazývá *single instruction multiple thread* případně *single program multiple data* (SIMT resp SPMD) a je navázán na TPC. Jedná se tedy o podobný přístup jako SIMD jádra uvnitř RV770 od ATI.

Podrobnější popis architektury GT200 naleznete v [19], [20], [25], [30], [31] a [32], detailní vysvětlení rozdílu SIMT proti SIMD v [33].

c) porovnání

Výpočetní jednotky ATI jsou méně komplexní než ty od nVidia a běhají na nižších frekvencích, na čip se jich ovšem vejde daleko více. Zjednodušeně se dá říci, že nVidia se vydala cestou zvětšování frekvence SP jednotek, zatímco ATI se snaží využít VLIW. Pro účely GPGPU vypadá vhodněji přístup ATI, která při počítání ve dvojité přesnosti (FP64) spojuje jednotlivá ALU, zatímco nVidia musí využít specializovanou jednotku.



Obr 11: zpracování instrukcí ATI vs nVidia (převzato z [19])

Z obrázku 11 je patrné, že zatímco nVidia má vždy své jedno SP využito, u ATI nastává situace, kdy nevyužijeme všech 5 ALU. Tento problém ale nebude tak markantní v GPGPU výpočtech, obzvláště při použití FP64, kdy je vytíženost ALU vyšší než v FP32.

2.7. reálná čísla v grafických kartách

Reálná čísla se ve výpočetní technice musí namapovat na obor celých čísel, čímž může dojít ke ztrátě informace. Je více způsobů, jak reálná čísla reprezentovat a jak s nimi pracovat, nejrozšířenější z nich je popsán standardem IEEE 754. Tento standard dodržuje většina procesorů, včetně těch od Intelu a AMD.

Standard IEEE 754 definuje aritmetiku v plovoucí čárce (FP) a to ve dvou reprezentacích, binární a decimální. Podle [7] popisuje norma tyto konkrétní situace:

- základní a rozšířený formát uložení numerických hodnot
- způsob provádění numerických operací: sčítání, odečítání, násobení, dělení, zbytek po dělení, druhá odmocnina, porovnání
- pravidla konverze mezi celočíselnými formáty a formáty s plovoucí řádovou čárkou
- konverze mezi různými formáty s plovoucí řádovou čárkou
- konverzi základního formátu s plovoucí řádovou čárkou na řetězec číslic
- práce s hodnotami *NaN* a výjimkami

Formát jednoduché přesnosti (FP32), podporovaný většinou grafických karet a procesorů, zabírá 32 bitů. První bit *S* značí znaménko, následuje 8 bitů *exp* pro exponent a 23 bitů *M* vyjadřujících mantisu. Při použití dvojité přesnosti (FP64) je vyhrazen jeden bit pro znaménko, 11 bitů pro exponent a 52 pro mantisu. Pro zrychlení výpočtů a přesnosti se mantisa ukládá převážně v *normalizovaném tvaru*, to znamená, že se do mantisy ukládají pouze hodnoty <1 ; 2-ε >. První bit je vždy 0, není tedy zapotřebí jej ukládat. Převod z vnitřní binární reprezentace na hodnotu vypadá takto:

$$X = (-1)^S \cdot 2^{(\exp - 127)} \cdot (1.\mathrm{M})_2 \quad \text{kde} \quad M = m_{22}^{-1} + m_{21}^{-2} + \ldots + m_1^{-22} + m_0^{-23} \quad \text{a} \quad \varepsilon = 2^{-23}$$

Pro uložení velmi malých čísel která nelze uložit normalizovaně se používá *denormalizovaný tvar*. Při použití tohoto tvaru je *exp* roven 0 a v *mantise* je uložena celá hodnota.

Specificky je uložena nula, nekonečno a NaN (*not a number*). Nula je reprezentována nulovým exponentem a nulovou mantisou. Nekonečno je reprezentováno maximálním exponentem a nulovou mantisou. Reprezentace NaN má maximální exponent a nenulovou mantisu, kde mantisa může obsahovat důvod vzniku NaN.

Při násobení se mantisy vynásobí a exponenty sečtou, při dělení se mantisy podělí a exponenty odečtou. Náročnější je sčítání a odečítání, kdy je nejprve potřeba upravit

exponenty na stejnou hodnotu a k tomu adekvátně posunout bity mantisy. Teprve poté se s mantisami provede požadovaná operace. Při posunu bitů mantisy dochází ke ztrátě přesnosti a může tedy dojít ke ztrátě informace. K největším nepřesnostem tak dochází právě při sčítání nebo odečítání čísel s rozdílnými exponenty. Nezanedbatelná nepřesnost také vzniká při odečítání čísel podobné velikosti, kdy výsledek je číslo blízké nule. Přesnost výsledného čísla je pak mnohem menší než přesnost odečítaných čísel.

Norma IEEE 754 kromě zmíněné vnitřní reprezentace čísel definuje také zaokrouhlování. Lze si vybrat jeden ze čtyř způsobů: směrem k nule, ke kladnému nekonečnu, k zápornému nekonečnu, k nejbližšímu sudému reprezentovatelnému číslu.

Většina nových karet na trhu se chlubí podporou IEEE 754 ať již v jednoduché nebo ve dvojité přesnosti. Bohužel co se píše v prezentačních materiálech není vždy tak úplně pravda. Doposud se na trhu neobjevila grafická karta s plnou podporou IEEE 754, pouze se některé k tomuto standardu blíží více a jiné méně.

Pro potřeby grafiky plně dostačovala nízká přesnost výpočtů, například v DX9 byla možnost počítat v takzvané poloviční přesnosti FP16 nebo FP24. Čím větší přesnost, tím pomalejší výpočet a větší paměťová náročnost, grafické karty šly přirozeně cestou zvyšování rychlosti.

V materiálech [16] zabývajících se CUDA upozorňuje nVidia na odlišnosti svých posledních (GT200) karet od IEEE 754, většina z nich platí pouze pro výpočty v jednoduché přesnosti:

- nemožnost nastavení zaokrouhlovacího modu
- nedochytávání výjimek a nestandardní výsledky operací s NaN
- absolutní hodnoty a negace nerespektují pravidla s NaN
- sčítání a násobení je často spojeno do jedné instrukce (FMAD), která ořezává mezivýsledek násobení
- nestandardní algoritmus dělení a odmocniny
- pro sčítání a násobení neumožňuje zaokrouhlování k +- nekonečnu
- nenormalizovaná čísla nejsou podporována a jsou oříznuta na nulu ještě před vlastní operací
- výsledek podtečení je nula

Je zřejmé, že výsledky určitých operací na GPU se mohou proti CPU lišit. Starší karty mají rozdílů v přesnosti ještě daleko více. V shader modelu 1 (DirectX8) se dokonce nepoužívala *plovoucí desetinná čárka* a na vše musela stačit reprezentace v *pevné řádové*

čárce (FX). Tato reprezentace se stále používá např. pro výpočty mlhy, kde nižší přesnost a vyšší výpočetní rychlost *pevné řádové čárky* plně vyhovuje. Na platformě x86 je možné tento formát použít skrze instrukce MMX.

Numerické hodnoty zapsané ve formátu *pevné řádové binární čárky* se chápou jako podmnožina racionálních čísel, jejichž hodnoty lze vyjádřit vztahem:

$$X = a/b \quad \text{kde} \quad a \in \mathbb{Z} \qquad b = 2^k \qquad k \in \mathbb{Z}^+$$

Protože je *b* celočíselnou mocninou dvojky, určuje jeho hodnota polohu binární čárky. Každé takto reprezentované číslo má řádovou binární čárku na stejné pozici, není tedy třeba ji v reprezentaci uvádět. Mezi výhody této reprezentace patří dopředu známá přesnost a jednodušší hardwarová implementace. Hlavní nevýhoda FX se projeví při velké dynamice výpočtů, tedy v momentě, kdy je velký poměr mezi nejvyšší a nejnižší absolutní hodnotou použitou při výpočtu. Pro zachování přesnosti by se v takovém případě musel v FX reprezentaci alokovat větší paměťový prostor oproti FP výpočtu.

3. Bez softwaru to nepůjde

3.1. co je to DirectX

DirectX je skupina programových rozhraní (API) určených k programování grafických a multimediálních aplikací pod MicroSoft Windows. Podle [10] a [34] byla hlavním důvodem vzniku DirectX potřeba MicroSoftu (MS) přesvědčit herní vývojáře k přechodu z DOSu na Windows 95. Pro tvorbu počítačových her nebyly původní Windows vůbec ideální. Na rozdíl od DOSu totiž neumožňovaly přímý přístup k hardwaru (grafické kartě, zvukové kartě, klávesnici, myši atd), ve Windows byly tyto záležitosti složitější a především znatelně pomalejší. Proto bylo v MS rozhodnuto vytvořit rozhraní pro rychlý přístup k hardwaru, které zároveň zachová hardwarovou nezávislost. Kromě 3D akcelerace je DirectX také nadstavbou ke zvukové kartě, 2D grafice, herním ovladačům atd.

V podstatě jediným velkým konkurentem pro DX ve 3D oblasti je rozhraní OpenGL. To bylo dříve (počátkem 90tých let) využíváno v CAD aplikacích a 3D animačních programech a vyžadovalo drahé, profesionální grafické karty. S rozmachem 3D akcelerátorů se stalo použitelné i pro vývoj počítačových her. Jeho obrovskou výhodou je nezávislost na operačním systému a jednodušší použitelnost. Zatímco v 90tých letech OpenGL pro 3D hry a aplikace jednoznačně vedlo, dnes můžeme říci, že v herním průmyslu je DX jasným vítězem. Důvodem je mnohem rychlejší vývoj DX zaměřený převážně právě do zábavního sektoru. Při vytváření specifikace DX10 již MicroSoft otevřeně spolupracoval s předními výrobci grafických karet, tedy ATI a nVidia.

Od DirectX8 (rok 2000) se začíná vyvíjet *shader model* a v DX9 se představuje první verze *HLSL* (high level shader language). Shader model definuje požadavky na shader procesory grafické karty. Jeden z vyšších jazyků, určených k jejich programování, je pak právě HLSL. V této práci otestujeme shader model 2.0 a relativně nový model 4.0, používaný od DX10 a Windows Vista.

V dalším textu budou popsány příkazy a techniky z DX9 a DX10 potřebné ke GPGPU výpočtům. Kvalitně zpracovanou dokumentaci a tutoriály k DirectX a HLSL naleznete na stránkách MicroSoftu [2].

3.2. trocha GPGPU teorie

Při programování obecných výpočtů na grafické kartě je potřeba si uvědomit některá omezení. Jednotky GPU jsou proudové procesory zpracovávající paralelně jeden kód (*kernel*) nad proudem vstupních dat. Spuštěná instance kódu se nazývá *thread*. Jelikož jednotlivé jednotky pracují nezávisle, nemohou mezi sebou sdílet data. Každý element dat je pouze přečtený ze vstupu jednotky, zpracován kernelem a poslán do další části pipeline. Není tedy možné mít jedna data zároveň pro čtení i zápis. Ideální GPGPU aplikace tedy má velký objem dat, na který aplikujeme jeden kód bez závislosti mezi daty. Pro zpětné využití vypočtených dat k dalším výpočtům je použita textura, která je po dokončení zápisu přemapována pro čtení. Procesu střídavého čtení a zápisu do stejné textury se říká *ping-pong*.

Jako proud dat použijeme 2D grid na který namapujeme texturu. Přesným namapováním vznikne pro každý pixel jeden fragment. Textura nám tedy nahrazuje paměťový prostor a souřadnice v textuře pak můžeme chápat jako paměťové adresy.

Kódu pro shader procesor se říká *kernel*. Ten si lze v analogii ke klasickému CPU programování představit jako tělo smyčky. Smyčka funguje jako podavač dat a kernel pak definuje jen operace nad proudem.

Za základní programovací techniky GPU lze považovat mapování, redukci, filtrování, práci s pamětí, třídění a vyhledávání. *Mapováním* se rozumí jen aplikace kernelu na všechny elementy v proudu. Pod *redukcí* chápeme zmenšení proudu, např. na jednu položku (ve které je výsledek operace nad ostatními elementy). *Filtrování* je typ redukce – odstranění některých elementů podle zadaného kriteria. Klasické algoritmy *třídění* nelze v GPU praxi využít kvůli datové závislosti, často se implementuje třídícími sítěmi. *Hledáním* jednoho prvku pomocí GPU nedosáhneme zrychlení, můžeme však v proudu vyhledávat více různých prvků zároveň.

Zápisu do paměti říkáme v GPU terminologii *scatter*. Mluvíme o něm v souvislosti s vertex procesorem, kde můžeme měnit pozici vrcholu. To z GPGPU hlediska znamená změnu indexu v poli. *Gather* je pak nazýváno čtení z paměti. To je možné v pixel shaderu, kde můžeme číst z textury libovolný pixel.

Pokročilé techniky pro zpracování numerických výpočtů na grafické kartě popisuje nVidia v [16].

3.3. DirectX9 konkrétně pro GPGPU

Příklad využití DX9 k obecným výpočtům je např. v [6], zároveň doporučuji prostudovat tutoriály DX9 od Microsoftu (viz [2]).

-v prvopočátku musíme vytvořit okno, bez něj DX nepojede. Okno musí být velikosti minimálně 32 x 32.

```
CreateWindow(...);
```

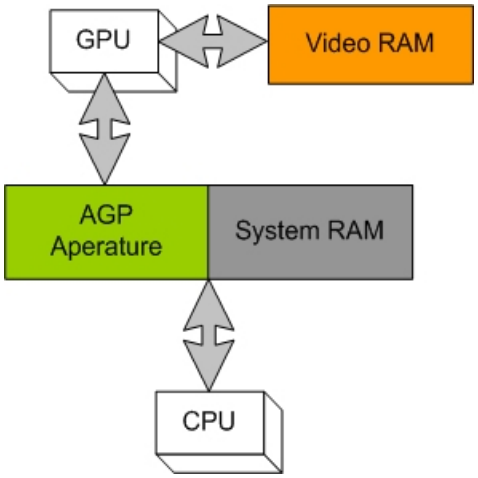
-inicializace DX s odkazem na vytvořené okno

```
LPDIRECT3D9 pD3D;
pD3D = Direct3DCreate9 (D3D_SDK_VERSION);
pD3D->CreateDevice(D3DADAPTER_DEFAULT,D3DDEVTYPE_HAL, hWnd,
D3DCREATE_HARDWARE_VERTEXPROCESSING, &parameters, &pD3DDev)));
```

-vytvoření textur se kterými budeme pracovat. *pTex* je určena pro vstupní data, do *pRenderTex* budeme renderovat výsledek a z *pOutputTex* budeme výsledek číst. Důležitý je parametr USAGE.

```
LPDIRECT3DTEXTURE9 pTex, pRenderTex, pOutputTex;
D3DXCreateTexture (pD3DDev, width, height, 1, 0, D3DFMT_R32F, D3DPOOL_MANAGED,
&pTex);
D3DXCreateTexture (pD3DDev, width, height, 1, D3DUSAGE_RENDERTARGET,
D3DFMT_R32F,  D3DPOOL_DEFAULT, &pRenderTex);
D3DXCreateTexture (pD3DDev, width, height, 1, 0, D3DFMT_R32F, D3DPOOL_SYSTEMMEM,
&pOutputTex);
```

Při manipulaci s texturami je potřeba uvědomit si způsob jejich ukládání. Dokumentace k DirectX obsahuje tento vysvětlující obrázek. Z něj je pochopitelné, že CPU nemá přístup do rychlé video paměti a že pro komunikaci mezi CPU a GPU je potřeba použít systémovou paměť.



Obr 12: přístup k paměti pod DirectX (převzato z [2])

-kopírování ze systémové paměti do video paměti

```
D3DXCreateTexture (pD3DDev, width, height, 1, 0, D3DFMT_R32F, D3DPOOL_SYSTEMMEM, &pReadTex);
D3DXCreateTexture (pD3DDev, width, height, 1, D3DUSAGE_RENDERTARGET, D3DFMT_R32F, D3DPOOL_DEFAULT, &pInTex);
pD3DDev->UpdateTexture (pReadTex, pInTex)));
```

-kopírování z video paměti do systémové paměti. Zdrojová textura musí být definována jako render target (parametr D3DUSAGE_RENDERTARGET při vytváření)

```
pReadTex->GetSurfaceLevel (0, &pReadSurf);
pIn Tex->GetSurfaceLevel (0, &pInSurf);
pD3DDev->GetRenderTargetData (pInSurf, pReadSurf)));
```

-při operacích čtení a zápisu musí být objekt (textura, buffer) uzamknut. K textuře se pak přistupuje přes obdélník alokovaný v paměti. Ten nemá rozměry textury, ale na konci každého řádku jsou uložena doplňková data (např. cache). Pro posun indexu na další řádek je tedy potřeba mít vypočtenou šířku obdélníku, zde proměnná *pitch.*

```
D3DLOCKED_RECT rect;
pOutputTex->LockRect (0, &rect, NULL, 0);
inMatrix = (float*) rect.pBits;
pitch = rect.Pitch/4;
inMatrix[i]=val;
pOutputTex->UnlockRect (0);
```

-kopírování dat mezi texturami a nastavení textury pro render se provádí přes povrchy

```
pOutputTex->GetSurfaceLevel (0, &pSurf);
pD3DDev->SetRenderTarget (0, pSurf);
pD3DDev->GetRenderTargetData (pSurfDest, pSurf);
```

-vytvoříme effect načtením kódu shaderu v HLSL (viz kap.3.4) ze souboru

```
D3DXCreateEffectFromFile(pD3DDev, "gpgpu.fx", NULL, NULL, 0, NULL, &pEffect, &pEB));
```

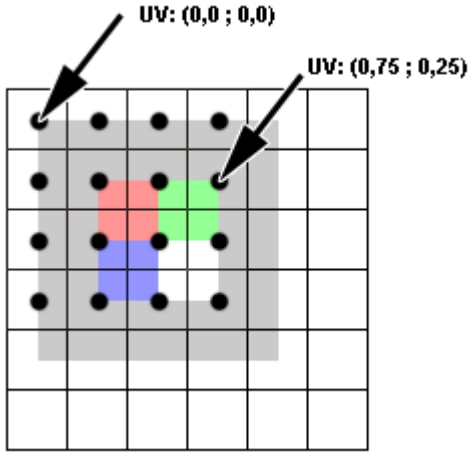
-do shaderů můžeme zaslat hodnotu, její přečtení viz kap.3.4

```
D3DXHANDLE hTexcoord2;
hTexcoord2 = pEffect->GetParameterBySemantic(NULL, "COLOR1");
pEffect->SetVector (hTexcoord2,&(D3DXVECTOR4 ( 1, 2, 3, 4 ))))
```

Menší nepříjemnost nastává při mapování obrazových bodů *(pixelů)* na souřadnice textury *(texely)*. V našem případě potřebujeme mít texely z textury namapované přesně na pixely obrazového bufferu. Podívejme se, jak ve [2] popisuje MicroSoft systém souřadnic použitý v DX9.

Display se skládá z pixelů (síť čtverců na obr.13) jejichž souřadnice se počítají ze středu pixelu (na obr.13 černé body). Naproti tomu má souřadnice textury svůj počátek v levém horním rohu. Oba systémy jsou tedy navzájem posunuté o polovinu pixelu.

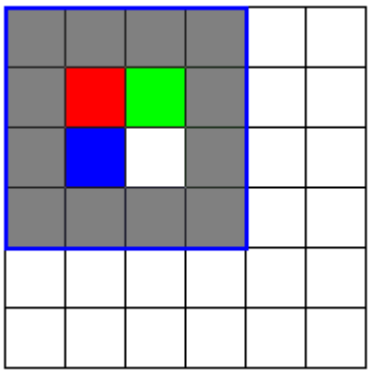
Pokud bychom intuitivně zarovnali levý horní roh textury se souřadnicí [0,0], dostaly by se čtyři texely na rozhraní jednoho pixelu. Který texel by byl nakonec kódem shaderu přečten, by byla sázka do loterie.



Obr 13: DX9 - mapování pixelů na texely

(převzato z [2])

Řešením je namapovat texturu posunutou právě o 0,5. V příkladu na obr.14 je levý horní a pravý spodní roh textury o velikosti 4 namapován na [-0,5; -0,5] a [3,5; 3,5].



Obr 14: DX9 - správné mapování pixelů na texely

(převzato z [2])

-nadefinujeme tedy obdélník posunutý o půl pixelu, na který namapujeme vrcholy textury. Bohužel při použití nenormalizovaných souřadnic se budeme muset spokojit s fixním vertex procesorem.

```
float fWidth = (float)( width ) - 0.5f;
float fHeight = (float)( height ) - 0.5f;

CUVERTEX myVertices[] =
{
        { -0.5f, -0.5f,             0.0f, 1.0f, 0.0f, 0.0f},
        { fWidth, -0.5f,            0.0f, 1.0f, 1.0f, 0.0f},
        { -0.5f,  fHeight,          0.0f, 1.0f, 0.0f, 1.0f},
        { fWidth, fHeight,          0.0f, 1.0f, 1.0f, 1.0f}
};
```

-dále musíme nadeklarovat, co ona čísla v našem obdélníku znamenají

```
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW | D3DFVF_TEX1)

typedef struct _CUVERTEX
{
        float x, y, z, w;
        float u, v;
} CUVERTEX;
```

-v renderovací smyčce přiřadíme shaderům textury a zvolíme, kterou techniku z efektu použijeme

```
pEffect->SetTexture ("mytex", pTex);
pEffect->SetTechnique ("pocty");
```

-v každém průchodu efektem popíšeme deklaraci vertex bufferu a vykreslíme ho

```
pEffect->Begin (&totalPasses, 0);
pEffect->BeginPass(pass);
pD3DDev->SetFVF(D3DFVF_CUSTOMVERTEX);
pD3DDev->DrawPrimitiveUP(D3DPT_TRIANGLESTRIP, 2, myVertices, sizeof(CUVERTEX));
pEffect->EndPass ();
pEffect->End ();
```

-renderovací část je uvozena příkazy

```
pD3DDev->BeginScene();
pD3DDev->EndScene();
```

3.4. shader 2.0 (DX9)

Základní popis HLSL pro verzi shaderu 2.0 je v [2].

-k texturám se přistupuje přes *sampler*. V definici sampleru říkáme, jak se k textuře chovat. Nám stačí říct, že nebudeme používat mipmapy.

```
texture mytex;
sampler2D mySamp = sampler_state
{
	Texture = (mytex);
	MipFilter = NONE;
};
```

-stejně jako v jazyce C můžeme definovat struktury. Ty se nám hodí pro vstupy a výstupy shaderů

```
struct VS_OUTPUT
{
	float4 Pos: POSITION;
	float2 Tex0: TEXCOORD0;
};
```

-kód vertex nebo pixel shaderu jsou jen jednoduché operace se vstupními daty

```
VS_OUTPUT VS (VS_INPUT In)
{
	VS_OUTPUT Out;
	Out.Pos = float4(In.Position,1);
	Out.Tex0 = In.TexCoord;
	return Out;
}
```

-můžeme si nechat zaslat proměnnou z kódu DirectX a nastavit ji jako globální proměnnou

```
uniform float4 gVar : COLOR1;
```

-na závěr definujeme techniku. V ní určíme, který shader použijeme a v jaké verzi ho chceme kompilovat

```
technique pocty
{
	pass P0
	{
		VertexShader = compile vs_2_0 VS ();
		PixelShader = compile ps_2_0 PS ();
	}
}
```

3.5. stejný příklad v DirectX10

Opět zde vycházím z manuálových stránek MicroSoftu [2] a hlavně z výborného vysvětlení DX10 v [18] a [15]. Způsob práce s DX10 je oproti DX9 úplně jiný, ke stejnému výsledku musíme napsat více kódu, který je ovšem transparentnější.

-před inicializací DX10 je nutné vytvořit okno o minimální velikosti 64 x 64

```
CreateWindow(...);
ID3D10Device * pD3DDev;
D3D10CreateDevice (NULL , D3D10_DRIVER_TYPE_HARDWARE, NULL , 0,
D3D10_SDK_VERSION, &pD3DDev);
```

-vytvoření efektu z HLSL kódu (kapitola 3.6)

```
ID3D10Effect * pEffect;
D3DX10CreateEffectFromFile( "gpgpugs02_DX10.fx", NULL, NULL, "fx_4_0", 0, 0, pD3DDev,
NULL, NULL, &pEffect, &pErrors, NULL );
```

-z vytvořeného efektu vybereme potřebné techniky (kód shaderu)

```
ID3D10EffectTechnique * pTechnique;
pTechnique = pEffect->GetTechniqueByName( "pocty" );
```

-do textury *pTexPong* budeme renderovat výsledek, k tomu je potřeba kromě textury vytvořit i renderovací "pohled" *pRTViewPong*

```
D3D10_TEXTURE2D_DESC descPong;
ZeroMemory( &descPong, sizeof(descPong) );
descPong.Width = width;
descPong.Height = height;
descPong.MipLevels = 1;
descPong.ArraySize = 1;
descPong.Format = DXGI_FORMAT_R32_FLOAT;
descPong.SampleDesc.Count = 1;
descPong.Usage = D3D10_USAGE_DEFAULT;
descPong.BindFlags = D3D10_BIND_RENDER_TARGET | D3D10_BIND_SHADER_RESOURCE;
pD3DDev->CreateTexture2D(&descPong,NULL,&pTexPong);

D3D10_RENDER_TARGET_VIEW_DESC descRT;
descRT.Format = descPong.Format;
descRT.ViewDimension = D3D10_RTV_DIMENSION_TEXTURE2D;
descRT.Texture2DArray.MipSlice = 0;
pD3DDev->CreateRenderTargetView( pTexPong, &descRT, &pRTViewPong );
```

-pokud tuto texturu budeme chtít použít zároveň pro vstup do shaderu, musíme vytvořit další, tentokrát zdrojový "pohled" *pSRViewPong*

```
D3D10_SHADER_RESOURCE_VIEW_DESC srDesc;
srDesc.Format = descPong.Format;
srDesc.ViewDimension = D3D10_SRV_DIMENSION_TEXTURE2D;
srDesc.Texture2D.MostDetailedMip = 0;
srDesc.Texture2D.MipLevels = 1;

pD3DDev->CreateShaderResourceView( pTexPong, &srDesc, &pSRViewPong );
```

-takto lze nastavit texturu pro vstup. Viz kap.3.6.

```
ID3D10EffectShaderResourceVariable * pSRVar;
pSRVar = pEffect->GetVariableByName("mytex")->AsShaderResource();
pSRVarTexIn->SetResource(pSRViewPong);
```

-a takto pro výstup

```
pD3DDev->OMSetRenderTargets(1, &pRTViewPong, NULL);
```

-k zaslání proměnné do shaderu stačí znát její název v HLSL, viz kap.3.6.

```
ID3D10EffectScalarVariable * pRVar;
pRVar = pEffect->GetVariableByName("pivot")->AsScalar();
pRVar->SetFloat( 1 );
```

-stejně jako v DX9 nemůžeme přistoupit přímo do video paměti, musíme vstupní data zapsat do textury v systémové paměti a tu pak překopírovat do grafické. Kde se bude textura nacházet určuje parametr Usage.

```
D3D10_TEXTURE2D_DESC desc;
desc.Width = width;
desc.Height = height;
desc.MipLevels = desc.ArraySize = 1;
desc.Format = DXGI_FORMAT_R32_FLOAT;
desc.SampleDesc.Count = 1;
desc.SampleDesc.Quality = 0;
desc.MiscFlags = 0;
desc.Usage = D3D10_USAGE_DYNAMIC;
desc.BindFlags = D3D10_BIND_SHADER_RESOURCE;
desc.CPUAccessFlags = D3D10_CPU_ACCESS_WRITE;

D3D10_TEXTURE2D_DESC descIn = desc;
descIn.Usage = D3D10_USAGE_DEFAULT;
descIn.BindFlags = D3D10_BIND_RENDER_TARGET | D3D10_BIND_SHADER_RESOURCE;
descIn.CPUAccessFlags = 0;

pD3DDev->CreateTexture2D( &desc, NULL, &pTexRel );
pD3DDev->CreateTexture2D( &descIn, NULL, &pTexIn );
```

-pro povolení zápisu do textury je potřeba ji nejprve namapovat

```
D3D10_MAPPED_TEXTURE2D mappedTex;
pTexRel->Map( D3D10CalcSubresource(0, 0, 1), D3D10_MAP_WRITE_DISCARD, 0, &mappedTex );
float* pTexels = (float*)mappedTex.pData;
DWORD pitch = mappedTex.RowPitch/4;
pTexels[i] = val;
pTexRel->Unmap( D3D10CalcSubresource(0, 0, 1) ); // Invalidate pointer to the resource
```

-po dokončení práce se vstupní texturou ji nakopírujeme do video paměti

```
pD3DDev->CopyResource( pTexIn, pTexRel );
```

-narozdíl od DX9 nemusíme při mapování texturu posouvat o půl pixelu, nepořebujeme tedy znát její velikost a namapujeme ji v normalizovaných souřadnicích, více viz [2]. Je tu ale jiný zádrhel, Y-ová souřadnice *rendertargetu* míří dolů oproti *viewportu* obstarávajícímu namapování vrcholů. Obejdeme to tak, že texturu na vrcholy namapujeme ozrcadlenou podle osy X.

```
CUVERTEX3 myVertices[] =
{
        {  1.0f,  1.0f, 0.0f,          1.0f, 0.0f },
        { -1.0f,  1.0f, 0.0f,          0.0f, 0.0f },
        {  1.0f, -1.0f, 0.0f,          1.0f, 1.0f },
        { -1.0f, -1.0f, 0.0f,          0.0f, 1.0f }
};
```

-vstupní jednotce pipeliny (viz obr.6) popíšeme layout posílaných dat

```
const D3D10_INPUT_ELEMENT_DESC defaultLayout[] =
{
        { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
D3D10_INPUT_PER_VERTEX_DATA, 0 },
        { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 12,
D3D10_INPUT_PER_VERTEX_DATA, 0 },
};

ID3D10InputLayout *  pVertexLayout;
D3D10_PASS_DESC passDesc;

pTechniqueSubMat->GetPassByIndex(0)->GetDesc(&passDesc);
pD3DDev->CreateInputLayout(defaultLayout, 2, passDesc.pIAInputSignature,
passDesc.IAInputSignatureSize, &pVertexLayout);
pD3DDev->IASetInputLayout(pVertexLayout);
```

-abychom mohli poslat vrcholy do vertex shaderu, musíme připravit vertex buffer

```
ID3D10Buffer * pVertexBuffer;
D3D10_BUFFER_DESC bd;
bd.Usage = D3D10_USAGE_DEFAULT;
bd.ByteWidth = sizeof(myVertices);
bd.BindFlags = D3D10_BIND_VERTEX_BUFFER;
bd.CPUAccessFlags = 0;
bd.MiscFlags = 0;

D3D10_SUBRESOURCE_DATA initData;
ZeroMemory(&initData, sizeof(initData));
initData.pSysMem = myVertices;
pD3DDev->CreateBuffer(&bd, &initData, &pVertexBuffer);
```

-tento buffer napojíme na vstup pipeliny

```
pD3DDev->IASetVertexBuffers(0, 1, &pVertexBuffer, &strides, &offsets);
pD3DDev->IASetPrimitiveTopology(D3D10_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP);
```

-nyní můžeme spustit vybranou techniku a tím provést výpočet

```
pTechnique->GetPassByIndex(p)->Apply(0);
pD3DDev->Draw(4, 0);
```

3.6. shader 4.0 (DX10)

Jak funguje HLSL shader modelu 4 je pěkně popsáno v [18] a samozřejmě opět na stránkách MS [2]. Oproti shader modelu 2.0 je zde hlavní rozdíl v unifikovanosti shaderů, díky němu se kód trochu rozrostl.

-připravíme definice nastavení jednotlivých částí pipeline. V nich vypneme všechny vymoženosti které se ke GPGPU nehodí.

```
BlendState NoBlending
{
	AlphaToCoverageEnable = FALSE;
	BlendEnable[0] = FALSE;
};

DepthStencilState DisableDepth
{
	DepthEnable = FALSE;
	DepthWriteMask = ZERO;
};

RasterizerState DisableCulling
{
	FillMode = SOLID;
	CullMode = NONE;
	DepthClipEnable = FALSE;
};
```

-stejně jako v HLSL 2.0 je k texturám možno přistupovat přes *samplery*, nadefinujeme si sampler, který nebude texturu vůbec měnit

```
SamplerState samNothing
{
	Filter = MIN_MAG_MIP_POINT;
	AddressU = CLAMP;
	AddressV = CLAMP;
};
```

-můžeme si nechat zaslat konstantu z kódu DirectX. Bohužel na rozdíl od DX9 se jedná opravdu o konstantu jejíž hodnotu není možné měnit.

```
uniform float4 gVar : COLOR1;
```

-budeme chtít přistupovat k textuře přiřazené v kódu DX10 podle jejího jména

```
Texture2D mytexTemp;
```

-vstupy a výstupy ze shaderů je, stejně jako v DX9, možno definovat strukturou

```
struct VS_OUTPUT
{
	float4 Pos: SV_POSITION;
	float2 Tex: TEXCOORD0;
};
```

-definování kernelu shaderů se změnilo spíš jen sémantikou (např. záměna POSITION za SV_TARGET)

```
float4 PSRow ( VS_OUTPUT In ) : SV_TARGET
{
	float4 Out;
	Out = ( 0, 0, 0, 0 );
	return Out;
}
```

-k texturám nyní kromě přístupu přes sampler můžeme přistupovat přímo v nenormalizovaných souřadnicích

```
pVal = mytex.Sample( samNothing, float2(pivotX, pivotY) );
pVal = mytex.Load( int3(pivot, pivot, 0) );
```

-v definici techniky, kromě výběru kompilovaného kódu a verze shaderu, zadáme nastavení chování částí pipeline (viz začátek této kapitoly)

```
technique10 poctyRow
{
	pass P0
	{
		SetVertexShader( CompileShader( vs_4_0, VSCopy() ) );
		SetGeometryShader( NULL );
		SetPixelShader( CompileShader( ps_4_0, PSRow() ) );

		SetBlendState( NoBlending, float4( 0.0f, 0.0f, 0.0f, 0.0f ), 0xFFFFFFFF );
		SetDepthStencilState( DisableDepth, 0 );
		SetRasterizerState( DisableCulling );
	}
}
```

4. Gaussova eliminace

Jako příklad GPGPU byl pro tuto práci zvolen výpočet soustavy lineárních rovnic. Pro počítačové zpracování se podle [47] nabízí mnoho známých metod, zde byla pro svoji jednoduchost a názornost vybrána základní z nich, Gaussova eliminace.

4.1. popis metody

Uvažujme obecnou soustavu lineárních rovnic $A\vec{x}=\vec{b}$ kde **A** je dvojrozměrná matice o rozměru *n* x *n*. Vektor $\vec{b}$ je vektor *n* konstant na pravých stranách rovnic a $\vec{x}$ je vektor *n* neznámých. Spojením matice **A** a vektoru $\vec{b}$ vznikne tzv. *rozšířená matice*:

$$\begin{pmatrix} a_{11}, & a_{12}, & \cdots & a_{1n}, & b_1 \\ a_{21}, & a_{22}, & \cdots & a_{2n}, & b_2 \\ & & \vdots & & \\ a_{n1}, & a_{n2}, & \cdots & a_{nn}, & b_n \end{pmatrix}$$

Gaussova eliminační metoda se skládá ze dvou kroků [23]. První fáze se nazývá *přímý chod* a dochází v ní k postupné úpravě rozšířené matice tak, aby pod hlavní diagonálou zůstaly samé nuly a zároveň na diagonále byly nenulové prvky.

$$\begin{pmatrix} a_{11}^{(0)}, & a_{12}^{(0)}, & \cdots & a_{1n}^{(0)}, & b_1^{(0)} \\ 0, & a_{22}^{(1)}, & \cdots & a_{2n}^{(1)}, & b_2^{(1)} \\ & & \vdots & & \\ 0, & 0, & \cdots & a_{nn}^{(n-1)}, & b_n^{(n-1)} \end{pmatrix}$$

Členy matice jsou určeny:

$$a_{ij}^{(0)}=a_{ij} \quad , \quad b_{ij}^{(0)}=b_{ij} \quad \text{kde } i = 1, 2, ..., n$$

$$a_{kj}^{(i)}=a_{kj}^{(i-1)}-\frac{a_{ki}^{(i-1)}}{a_{ii}^{(i-1)}}\cdot a_{ij}^{(i-1)} \quad , \quad b_k^{(i)}=b_k^{(i-1)}-\frac{a_{kj}^{(i-1)}}{a_{ii}^{(i-1)}}\cdot b_i^{(i-1)}$$

kde $i = 1, 2, ..., n-1; k = i + 1, i + 2, ..., n; j = i + 1, i + 2, ..., n$

Druhé fázi se říká *zpětný chod*, z výsledné matice z první fáze se spočte vektor $\vec{x}$ podle vzorce:

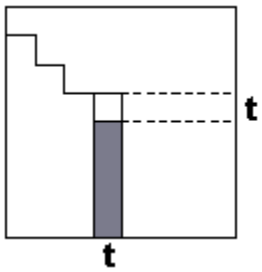
$$x_i=\frac{b_i^{(i-1)}-\sum_{j=i+1}^{n} a_{ij}^{(i-1)}\cdot x_j}{a_{ii}^{(i-1)}} \quad \text{kde } i = n, n-1, ..., 1$$

4.2. algoritmus pro CPU a GPU

Základní algoritmus pro první fázi Gaussovy eliminace vypadá na CPU takto:

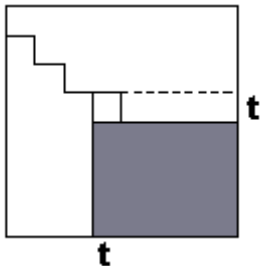
```
for (int t=0; t<height-1; t++) // pivot loop
        pivot = inMatrix[t][t];
        for (int i=(t+1); i<height; i++)  // row loop
                tmpMatrix[i] = inMatrix[t][i] / pivot;
        for (int i=(t+1); i<height; i++)
                for (int j=t; j<width; j++) // submat loop
                        inMatrix[j][i] = inMatrix[j][i] - tmpMatrix[i] * inMatrix[j][t];
```

Na GPU je převod již poměrně jednoduchý, těla obou vnitřních smyček (*row* a *submat*) jsou kernely pixel shaderu. Hlavní smyčka pro posun pivotu zůstane k obsluze CPU. Pro lepší představení si algoritmu, poslouží následující dva obrázky. Pivot je na pozici [**t, t**]. Hodnoty vpravo a dolů od pivotu je potřeba ještě spočítat, zbylé hodnoty jsou již finální.



Obr 15: Gaussova eliminace - sloupec (převzato z [11])

Ve smyčce *row* (prosím nepřekládat, ve skutečnosti jde o sloupec) pro pivot s indexem **t** počítáme vybarvené hodnoty tak, že aktuální hodnotu vydělíme pivotem.



Obr 16: Gaussova eliminace - submat (převzato z [11])

Následující smyčka *submat* pracuje s hodnotami spočtených v předchozí smyčce. Přepočítá celou vybarvenou submatici, čímž pod pivotem ve sloupci **t** vzniknou kýžené nuly. V algoritmu pro *zpětný chod* se hodnoty vektoru $\vec{x}$ přiřadí prvkům pole b.

```
for (int t=(height-1); t>=0; t++)
        for (int i=(t+1); i<height; i++)
                b[t] = b[t] – a[i][t] * b[i];
        b[t] = b[t] / a[t][t];
```

4.3. řešení pod DirectX9

Do textury přistupujeme přes souřadnice texelů. DirectX9 bohužel dovede používat pouze normalizované souřadnice v rozsahu (0, 1). Pokud tedy v průběhu shaderu potřebujeme sáhnout na určitý bod v textuře, musíme tento přístup také znormalizovat. K tomu potřebujeme znát rozměry textury. Ty nelze v průběhu *kernelu* zjistit, přeposíláme si je skrze DirectX takto:

```
D3DXHANDLE hTexcoord2 = pEffect->GetParameterBySemantic(NULL, "COLOR1");
pEffect->SetVector (hTexcoord2,&(D3DXVECTOR4 ((float)width,(float)height,256,(float)t)))))
```

Co je pEffect viz kapitoly 3.3 a 3.4. Tyto dva řádky pošlou do shader registru COLOR1 rozměry matice a aktuální pozici pivotu. Jedná se o čtyř-složkový vektor, kde první dvě komponenty (*r* a *g*) obsahují velikost matice a poslední (*a*) je hodnota pivotu. Tento registr spojíme s proměnnou:

```
uniform float4 pivotPos : COLOR1;  // pivotPos nabindujeme na registry COLOR1
```

Používám tři textury, do *tmp* ukládám právě spočítaný sloupec, z textury *in* čtu a do textury *pong* zapisuji výsledek. Na CPU zůstává smyčka posouvající pivot. V ní dochází k předání souřadnice pivotu a rozměrů textury shaderům a postupnému volání *row* a *submat* efektů . Postup, jak toho bylo docíleno popisuje kapitola 3.3. Na konci této hlavní smyčky dojde k prohození textur *in* a *pong* – střídavě pro zápis a čtení, jednoduše prohozením ukazatelů na texturu a její povrch.

Nejprve spočítáme sloupec s pivotem, jeho výstupem je jednorozměrná textura. Souřadnici pivotu znormalizujeme prostým vydělením pozice rozměrem textury.

```
PS_OUTPUT psRow (VS_OUTPUT In) : COLOR
// tmpMatrix[t+i*width] = inMatrix[t+i*width] / pivot;
{
	PS_OUTPUT Out;
	float pivot, inMat;
	float pivotX, pivotY;

	pivotX = pivotRaw.a / pivotRaw.r; // pivot / width
	pivotY = pivotRaw.a / pivotRaw.g; // pivot / height

	inMat = tex2D( mySamp, float2( pivotX, In.Tex0.y ) ).r; // inMatrix[t+i*width]
	pivot = tex2D( mySamp, float2( pivotX, pivotY ) ).r; // inMatrix[t+t*width]

	Out.Color.r = inMat / pivot; // tmpMatrix[t+i*width]

	return Out;
}
```

Tuto texturu použijeme na vstupu dalšího shaderu (proměnná *tmpMat*). Shader se provádí pro všechny pixely, proto abychom počítali jen submatici, je potřeba použít podmínku. Pokud není splněna, zůstane zachována původní (tedy již finální) hodnota.

```
PS_OUTPUT psSubMat (VS_OUTPUT In) : COLOR
// inMatrix[j+i*width] = inMatrix[j+i*width] - tmpMatrix[t+i*width] * inMatrix[j+t*width];
{
	PS_OUTPUT Out;
	float inMatA,tmpMat,inMatB;
	float pivotY,pivotYplus,pivotXminus;
	float tmp;

	pivotY=pivotRaw.a/pivotRaw.g; // pivot/height
	pivotXminus=(pivotRaw.a-0.5)/pivotRaw.r; // pivot/width
	pivotYplus=(pivotRaw.a+0.5)/pivotRaw.g; // pivot/height

	inMatA = tex2D(mySamp, In.Tex0).r; // inMatrix[j+i*width]
	inMatB = tex2D(mySamp, float2(In.Tex0.x, pivotY)).r; // inMatrix[j+t*width]
	tmpMat = tex1D(mySampTemp, In.Tex0.y).r; // tmpMatrix[t+i*width]

	// for (int i=(t+1); i<height; i++) for (int j=t; j<width; j++)
	if (In.Tex0.y>pivotYplus && In.Tex0.x>pivotXminus) // submat
	{
		tmp = tmpMat*inMatB;
		inMatA = inMatA-tmp;
	} else // original
		inMatA = tex2D(mySamp, In.Tex0).r;

	Out.Color.r = inMatA; // inMatrix[j+i*width]

	return Out;
}
```

4.4. úpravy pro DirectX10

Algoritmus je stejný, jako v předchozí kapitole, vyskytne se zde jen pár specifických úprav.

Protože není potřeba mapovat texturu posunutě o půl pixel (viz kap. 3.3), je nám k dispozici programovatelný vertex shader. V našem příkladě budeme data jen posílat dále.

```
VS_OUTPUT VSCopy( float4 Pos : POSITION , float2 Tex : TEXCOORD0)
{
	VS_OUTPUT Out;
	Out.Pos = Pos;
	Out.Tex = Tex;
	return Out;
}
```

Do textur je možno přistupovat nejen normalizovanými souřadnicemi skrze sampler, ale i přímo funkcí *load*. Výpočet sloupce s pivotem je možno vyřešit takto:

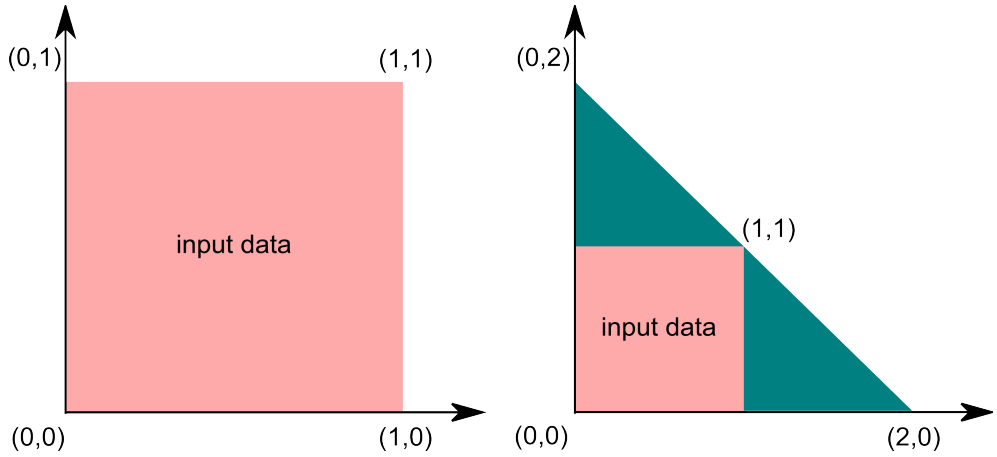
```
float4 PSRow ( VS_OUTPUT In ) : SV_TARGET
// tmpMatrix[t+i*width] = inMatrix[t+i*width] / pivot;
{
	float4 Out;
	float pVal,inMat;

	inMat = mytex.Load( int3(pivot, In.Tex.y*height, 0) ); // inMatrix[t+i*width]
	pVal = mytex.Load( int3(pivot, pivot, 0) ); // inMatrix[t+t*width]

	Out.r = inMat / pVal; // tmpMatrix[t+i*width]

	return Out;
}
```

Menších úprav doznal i *kernel* pro výpočet submatice:

```
float4 PSSubMat ( VS_OUTPUT In ) : SV_TARGET
// inMatrix[j+i*width] = inMatrix[j+i*width] - tmpMatrix[t+i*width] * inMatrix[j+t*width];
{
	float4 Out;
	float inMatA, tmpMat, inMatB, inMatABranch;

	inMatB = mytex.Load( int3(In.Tex.x*width, pivot, 0) );
	tmpMat = mytexTemp.Sample( samNothing, In.Tex.xy ); // tmpMatrix[t+i*width]

	inMatA = mytex.Sample( samNothing, In.Tex ); // inMatrix[j+i*width]
	inMatABranch = inMatA-tmpMat*inMatB;

	// for (int i=(t+1); i<height; i++) for (int j=t; j<width; j++)
	if (In.Tex.y*height>pivot && In.Tex.x*width>pivot) // submat
		inMatA = inMatABranch;

	Out.r = inMatA; // inMatrix[j+i*width]

	return Out;
}
```

4.5. optimalizace DirectX

Hlavní algoritmus je již doufám zřejmý, na závěr můžeme tedy přikročit k optimalizaci. Vraťme se k definici vertexu z kapitoly 3.3. Jde o obdélník, pro jednoduchost počítejme s vrcholy (0,0,0), (1,0,0), (1,1,0), (0,1,0), na nějž se namapuje textura s vrcholy (0,0), (1,0), (1,1), (0,1). Na grafické kartě se ale náš obdélník rozdělí na dva trojúhelníky. GPU při zpracovávání přednačítá texturu poblíž právě zpracovávaného texelu, ovšem jen z aktuálního trianglu. Při rozdělení naší textury tak dochází ke zbytečnému zahazování těchto přednačtených dat. Řešení je prosté. Místo obdélníku který se stejně interně rozdělí na dva trojúhelníky, nadefinujeme rovnou trojúhelník. Nastavíme mu rozměry (0,0,0), (2,0,0), (0,2,0) a do něj namapujeme texturu o souřadnicích (0,0), (2,0), (0,2). Na obr.17 vidíme výsledek, všechna data máme pěkně v jednom trianglu. Část pipeline, starající se o ořezávání neviditelných částí obrazu *(clipping)*, provede oříznutí na původní obdélník.



Obr 17: optimalizace mapováním textury

Jediná změna proti stávajícímu kódu je v předefinování vrcholů, zde ve verzi pro DX9, posunutou o polovinu pixelu:

```
CUVERTEX myVertices[] =
{
        { -0.5f, -0.5f, 0.0f,              1.0f, 0.0f, 0.0f},
        { (width-0.5)*2, -0.5f, 0.0f,      1.0f, 2.0f, 0.0f},
        { -0.5f, (height-05)*2, 0.0f,      1.0f, 0.0f, 2.0f}
}
```

Další možností optimalizace je v předpočítání některých hodnot pro pixel shader již ve vertex shaderu, který je v GPGPU výpočtech znatelně méně vytížen. V našem příkladě Gaussovy eliminace ale nenašel uplatnění.

5. souboj CPU vs GPU

5.1. způsob měření

Nejprve se podívejme na seznam hardwaru, použitého při testování.

CPU	GPU
AMD Phenom 9950 Quad-Core 2,6GHz	ATI Radeon HD 4870
AMD Turion 64 ML-32 1,8GHz	ATI mobility Radeon x700
AMD X2 Dual-Core 5000+ 2,6GHz	ATI Radeon HD 2600 XT
AMD X2 Dual-Core 5000+ 2,6GHz	nVidia GeForce 9600 GT
AMD X2 Dual-Core 5000+ 2,6GHz	nVidia GeForce GTX280
AMD X2 Dual-Core 5400+ 2,8GHz	ATI Radeon HD 3200
Intel Core2 E6850 3GHz	nVidia GeForce GTX260
Intel Core2 E8400 3GHz	nVidia Quadro FX 1500
Intel Core2 T7200 2GHz	nVidia GeForce Go 7600
Intel Core2 T9400 2,6GHz	nVidia GeForce 9600M GT
Intel Core2 6300 1,8GHz	nVidia GeForce 7600 GS
Intel Pentium 4 1,8GHz	ATI Radeon 9600

Tab 1: testovaný hardware

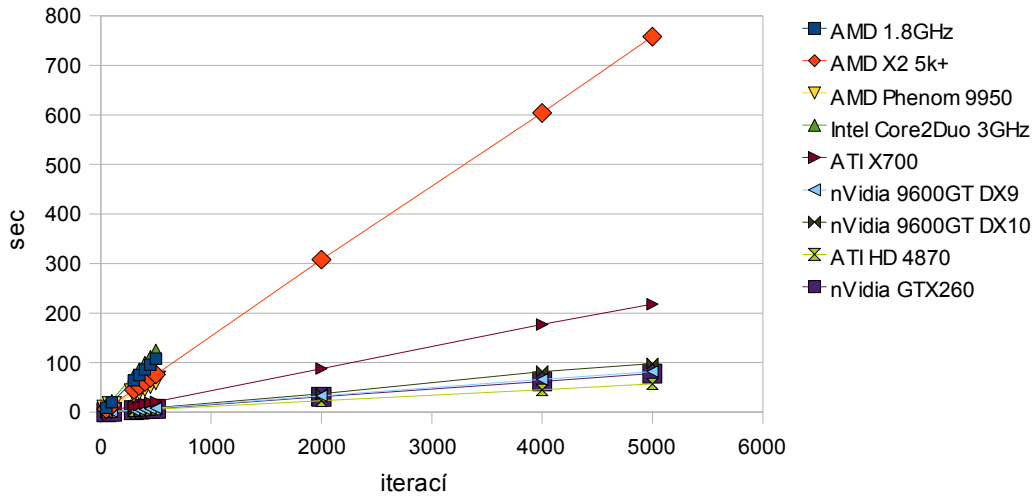
V kapitole 2.7 je probrána Gaussova eliminace, právě první fáze této metody byla použita pro veškeré testy. Řešená soustava rovnic je zadána formou matice. Kvůli zvýšení citlivosti měření, byla většina úloh počítána s několika iteracemi. Všechny matice použité pro měření rychlosti výpočtu obsahují náhodná čísla v rozsahu 0 - 1000 a jsou k nalezení na přiloženém CD *(inputs.zip)*. Jedná se o matice těchto velikostí *n*: 10, 50, 100, 200, 500, 1000, 1500, 2000, 3000, 4000. Počet prvků matice je $N = (n + 1) * n$.

Algoritmus počítaný na CPU neobsahoval žádné optimalizace a při výpočtu bylo použito jen jedno jádro procesoru. Pokud není uvedeno jinak, počítal procesor v jednoduché přesnosti. Stejně tak výpočet na GPU nebyl nijak optimalizován a pokud to karta povolovala, byl prováděn v jednoduché přesnosti. Naměřené časy neobsahují dobu potřebnou k načtení vstupních dat a zápisu dat výstupních. Doba výpočtu na grafické kartě je ovlivněna rychlostí procesoru, který se stále stará o hlavní smyčku posouvající pivot, tato skutečnost byla při měření zanedbána. V uvedených tabulkách jsou uvedeny pouze nejzajímavější hodnoty, veškeré naměřené výstupy jsou zaznamenány na CD *(outputs.zip)*.

5.2. tabulky a grafy

Takto byl měřen průměrný počet iterací za sekundu, není překvapením, že jde o lineární závislost. Finální poměr iterací za sekundu je spočítán z několika nejvyšších naměřených hodnot (v tabulce 2 uvedeny modře). Z důvody časové náročnosti a zřejmé linearity výsledků nebylo na několika CPU provedeno měření pro nejvyšší hodnoty iterací.

n=200		iterací										iter/sec
		50	100	300	350	400	450	500	2000	4000	5000	
CPU	AMD 1.8GHz	9	20	64	75	86	96	108				4,66
	AMD X2 5k+	6	14	45	53	61	69	76	308	604	758	6,57
	AMD X2 5k+ (FP64)	3	8	27	32	36	41	47	189	381	477	10,55
	AMD Phenom 9950	5	12	38	44	51	57	64				7,88
	Intel Core2Duo 3GHz	11	24	75	88	101	113	126				3,97
GPU	ATI X700	2	4	12	14	17	19	21	88	177	218	22,75
	nVidia 9600GT DX9	0	1	4	5	6	7	8	33	66	82	60,73
	nVidia 9600GT DX10	0	1	5	6	7	8	9	37	82	98	51,28
	ATI HD 4870	0	1	3	4	4	5	6	23	45	57	87,85
	nVidia GTX260	0	1	4	5	6	7	7	31	62	78	64,38

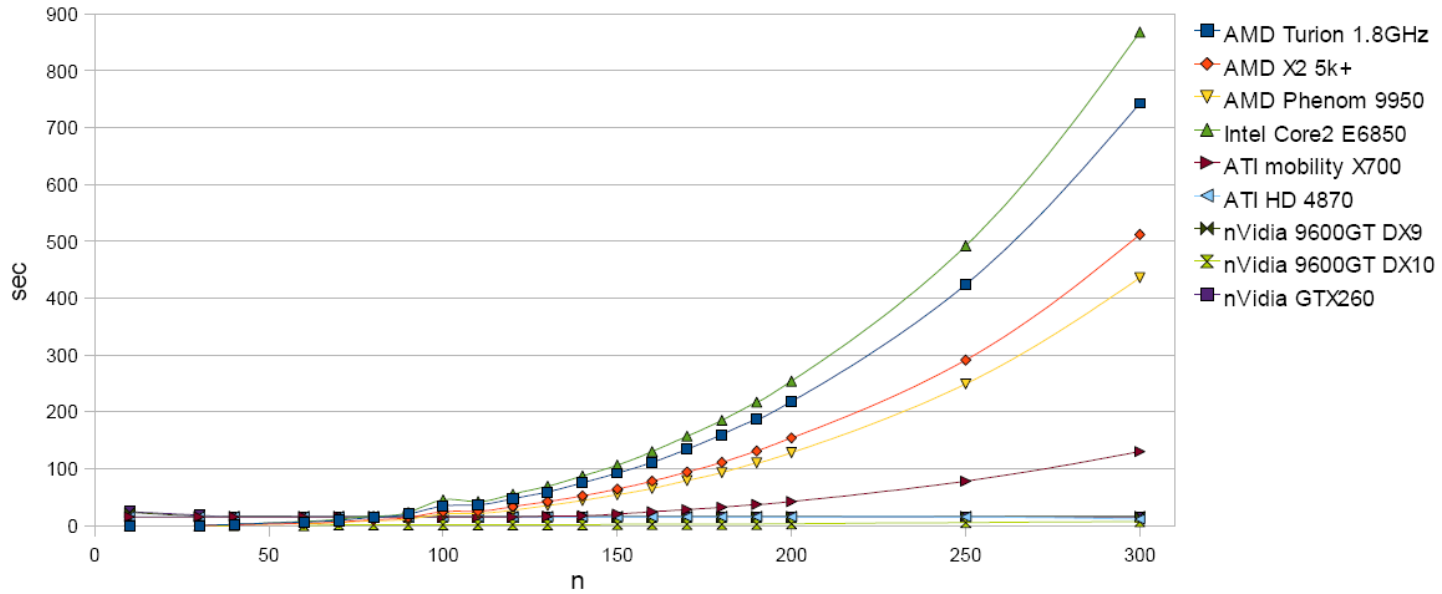


Tab 2: n = 200; i = var

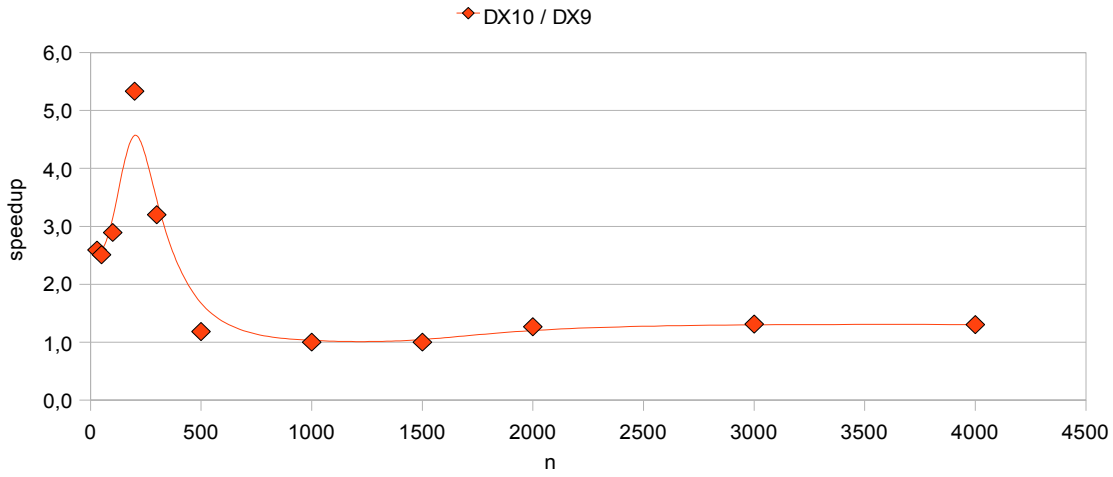
Na následujících stranách jsou grafy zajímavější. V tab.3 jde o naměřené časy výpočtu v závislosti na velikosti vstupní matice. Rozdíl v rychlostech mezi DX9 a DX10 ukazují tabulky 4 a 5. Pro nás největší výpovědní hodnotu mají poslední dva grafy 6 a 7 znázorňující zrychlení GPU proti CPU, tedy poměr dob výpočtu mezi grafickou kartou a nejrychlejším testovaným procesorem.

iter=1000		n																			
		10	30	40	60	70	80	90	100	110	120	130	140	150	160	170	180	190	200	250	300
CPU	AMD Turion 1.8GHz	0	0	2	6	9	14	20	34	36	47	59	75	92	111	134	160	187	218	424	743
CPU	AMD X2 5k+	0	0	1	4	6	10	14	24	25	33	42	52	64	78	94	111	131	154	291	512
CPU	AMD Phenom 9950	0	0	1	3	5	8	11	20	21	27	35	44	54	65	79	93	110	128	249	436
CPU	Intel Core2 E6850	0	0	2	7	11	16	23	45	42	55	69	87	106	130	157	185	217	254	492	868
GPU	ATI mobility X700	15	15	15	15	15	15	15	15	15	15	16	17	20	24	28	32	37	42	78	130
GPU	ATI HD 4870	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	11
GPU	nVidia 9600GT DX9	23	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
GPU	nVidia 9600GT DX10	0	0	0	0	1	1	1	1	1	1	1	1	2	2	2	2	2	3	5	7
GPU	nVidia GTX260	24	18	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15

Tab 3: iter = 1000; n = var



AMD X2 2,6GHz		n (iter=var)										
		30	50	100	200	300	500	1000	1500	2000	3000	4000
nVidia GTX280	DX9 Vista	1,6	1,6	1,6	1,6	1,6	1,9	9,5	31,0	63,3	210,0	486,7
	DX10 Vista	0,6	0,6	0,6	0,3	0,5	1,6	9,5	31,0	50,0	160,0	373,3
zrychlení	DX10 / DX9	2,6	2,5	2,9	5,3	3,2	1,2	1,0	1,0	1,3	1,3	1,3

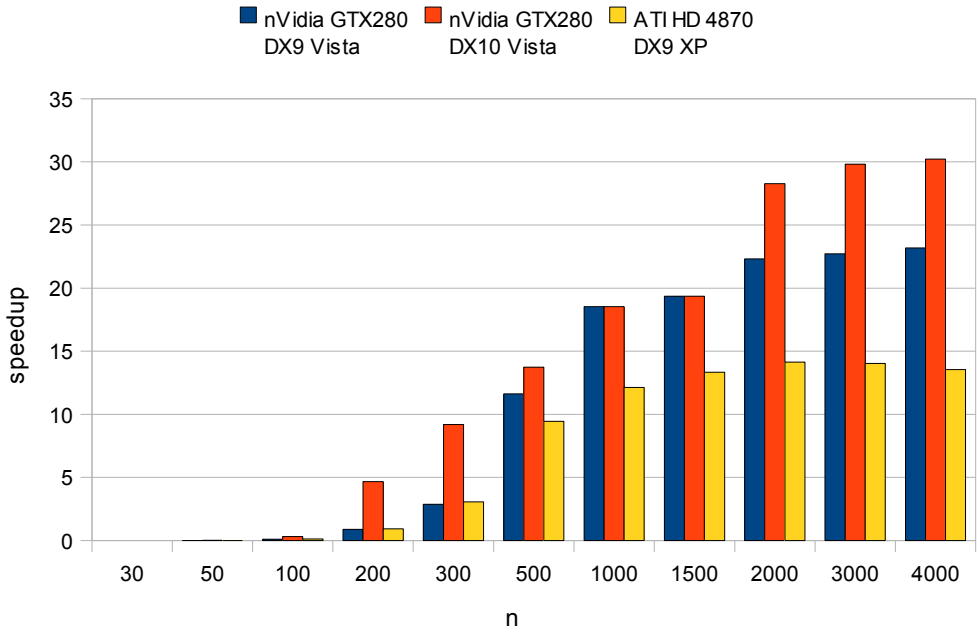


Tab 4: zrychlení DX10 vs DX9

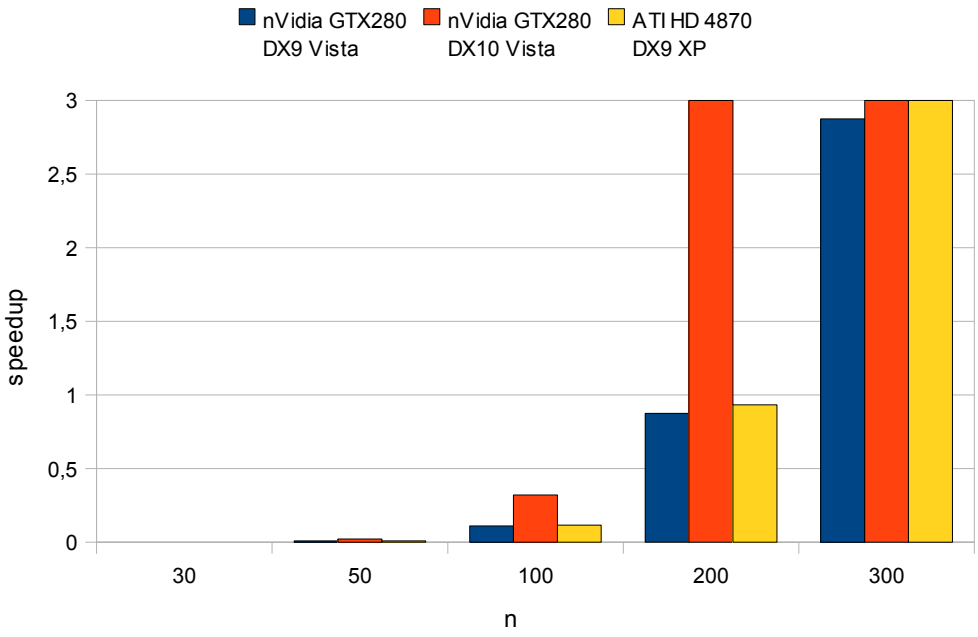
AMD Phenom 2,6GHz		n (iter=400)					
		50	100	200	500	700	1000
ATI Radeon HD 4870	DX9 XP	6	6	6	9	22	58
	DX9 Vista	6	6	6	9	21	57
zrychlení	XP / Vista	1	1	1	1	1,05	1,02

Tab 5: zrychlení Vista vs XP

Pro porovnání byl z hardwaru účastnícího se měření, vybrán nejrychlejší procesor *AMD Phenom 9950 Quad-Core 2,6GHz* proti dvěma nejvýkonnějším kartám od různých výrobců: *ATI Radeon HD 4870* a *nVidia GeForce GTX280*. Hodnoty ze kterých grafy 6 a 7 vznikly jsou uvedeny v tab.8.



Tab 6: zrychlení GPU vs CPU pro matice n = 30 - 4000



Tab 7: zrychlení GPU vs CPU pro matice n = 30 - 1000

Po přepočtení doby výpočtu na GFlops se k teoretickým hodnotám příliš nepřiblížíme, k tomu by bylo zapotřebí sestavit speciální benchmark. V použitém vzorci je zanedbán čas pro zpracování podmínky a přístupu do paměti, počítány jsou pouze aritmetické instrukce.

Těchto instrukcí je při výpočtu celé soustavy na CPU celkem použito:

*((2 * n) [ROW] + (4 * n^2) [SUBMAT]) * n [PIVOT] =* ***2 * n^2 + 4 * n^3***

Po přepsání algoritmu na DX9 je aritmetických operací více z důvodu přepočtů pro přístup na konkrétní místo v textuře:

5 * n^2 + 10 * n^3

Výpočty v DX10 jsou díky funkci *load* úspornější, stačilo by jen:

2 * n^2 + 5 * n^3

Abych mohl jednoduše porovnat výkon DX9 vůči DX10, používal jsem i při výpočtech na DX10 kód co nejvíce podobný předchozí verzi DirectX, počet aritmetických instrukcí je v mém měření tedy pro obě verze shodný.

V tabulce 8 jsou uvedeny průměrné časy výpočtu jedné iterace včetně přepočtu na počet operací za sekundu.

n	AMD Phenom Quad FP32 XP 32		nVidia GTX280 DX9 Vista 64		nVidia GTX280 DX10 Vista 64		ATI HD 4870 DX9 XP 32	
	msec	GFlops	msec	GFlops	msec	GFlops	msec	GFlops
30	0,00		16,29	0,02	6,29	0,04	15,57	0,02
50	0,14	3,54	16,14	0,08	6,43	0,20	15,57	0,08
100	1,80	2,23	16,20	0,62	5,60	1,79	15,60	0,64
200	14,00	2,29	16,00	5,01	3,00	26,73	15,00	5,35
300	46,00	2,35	16,00	16,90	5,00	54,09	15,00	18,03
500	215,71	2,32	18,57	67,38	15,71	79,63	22,86	54,74
1000	1760,00	2,27	95,00	105,32	95,00	105,32	145,00	69,00
1500	6000,00	2,25	310,00	108,91	310,00	108,91	450,00	75,03
2000	14133,33	2,26	633,33	126,35	500,00	160,04	1000,00	80,02
3000	47700,00	2,26	2100,00	128,59	1600,00	168,78	3400,00	79,43
4000	112800,00	2,27	4866,67	131,52	3733,33	171,45	8333,33	76,81

Tab 8: zrychlení a výkon v GFlops

5.3. přesnost, kapitola sama pro sebe

Doposud byla zmiňována především rychlost výpočtu, zajímavé je i porovnání přesnosti. Ta se totiž projevila jako hlavní slabina grafických karet použitých pro negrafické výpočty. Situace se začíná zlepšovat až nyní s rozmachem GPGPU, kdy grafické karty umožňují výpočty ve dvojité přesnosti FP64. Tuto přesnost nelze bohužel v testovaném rozhraní DirectX zapnout, nebylo tedy možno ji otestovat.

Podle kapitoly 2.7 je zřejmé, že výsledky CPU dodržujícího normu IEEE 754, se budou od těch na GPU lišit. Při výpočtech Gaussovo eliminací se některé hodnoty přepočítávají *n*-krát, podle postupu pivotu. Chyba zanesena na počátku se s každým dalším průchodem zvětšuje a zároveň se zanáší chyba nová. Každý čip se s přesností vypořádal trochu jinak, žádný však nedosahuje ani FP32 dle zmíněné normy.

V tabulce 9 jsou uvedeny výsledky podílu 3 / 6,5 spočteném ve dvou různých přesnostech k porovnání s GPU. Správný výsledek je perioda 3 / 6,5 = $0{,}461538\overline{461538}$

	3 / 6,5	0.0030 / 0.0065
přesná hodnota	0.461538461538461538	0.461538461538461538
FP64 CPU	0.46153846153846156	0.46153846153846156
FP32 CPU	0.46153846383094788	0.46153846383094788
FP32 nVidia GTX280; 9600M	0.46153849363327026	0.46153843402862549
FP32 nVidia Go 7600; FX 1500	0.46153849363327026	0.46153846383094788
FP32 ATI HD 2600; HD 4870	0.46153843402862549	0.46153843402862549
FP24 ATI x700	0.46153259277343750	0.46153259277343750

Tab 9: přesnost CPU vs GPU

Dělení na nejpřesnější testované grafické kartě je tedy v uvedeném případě přesné na 7 desetinných míst. V použitém algoritmu Gaussovy eliminace je kromě dělení použita i operace odečítaní, která chybu dále zvětšuje [47]. Například při odečítaní podobných čísel známých s přesností 7 desetinných míst a lišících se v posledních 3 místech je výsledek přesný na pouhé 3 místa. Malá ukázka: 0,4615384 – 0,4615227 = 0,0000157.

Tento problém se projevuje také na CPU, kde výsledky v jednoduché přesnosti jsou pro větší matice značně rozdílné od výsledků počítaných v přesnosti dvojité a od výsledků teoretických. Výkon 64-bitových procesorů se ale při dvojité přesnosti zmenší jen velmi málo v některých případech se dokonce zvýší.

Přesnost použitého algoritmu je možné zlepšit například metodou *pivotingu*, kdy se pivot vybere podle určitého kritéria, zpravidla prvek s největší absolutní hodnotou v řešeném řádku. V soustavě lineárních rovnic lze beztrestně přehazovat řádky a sloupce matice, tím lze na místo původního pivotu dostat libovolné jiné číslo z matice.

5.4. zhodnocení výsledků

Při řešení jsou viditelné rozdíly mezi grafickými kartami. Ty se liší především ve výpočetní přesnosti a v maximální velikosti textury. Nejhůře na tom jsou grafické karty osazované v noteboocích, které jsou navíc ještě znatelně pomalejší než jejich stolní verze. Ke GPGPU výpočtům je rozhodně nelze nedoporučit, ani pro malé velikosti matic. Například ATI x700 má udávanou maximální velikost textury 2096x2096, ale již při velikostech 2000x2000 se projevuje optimalizace karty a hodnoty čtené z konce vstupní textury jsou zkreslené.

Pod DX9 se síla grafiky projevuje až od určitého objemu dat, do té doby za procesorem zaostává, především kvůli pomalému ping-pongu. V testovaných úlohách nastával zlom kolem hodnoty n=200, tedy u 4000 prvků a $8 \cdot 10^6$ výpočtů. V DX10 je situace trochu jiná, k paměti se přistupuje jiným způsobem a některé grafické karty již nejsou výrazně pomalejší ani pro malé rozměry matic. Celkově vyhází DX10 na stejném hardwaru a operačním systému výkonnostně lépe. Implementace DX9 grafických karet ATI je pod Windows Vista pro GPGPU stejně výkonná, jako to samé rozhraní pod Windows XP.

Pokud porovnáme nejmodernější grafické čipy RV770 a GT200 použité v testovaných kartách nVidia GeForce GTX280 AMP! a ATI Radeon HD 4870, pak u použitého algoritmu vychází jednoznačně lépe nVidia. Každá z obou karet podává výsledky lehce odlišné od FP32, nVidia je však znatelně rychlejší. Téměř dvojnásobná ztráta výkonu ATI vůči nVidia je zřejmě způsobena vyhodnocováním podmínky v kernelu pixel shaderu. Potíž je v tom, že ATI má jen jednu jednotku k vyhodnocení skoku pro pět výpočetních jednotek. Tento problém se dá vyřešit úpravou algoritmu.

Vhodnost výpočtu Gaussovy eliminace na grafické kartě závisí především na vstupních datech, použité kartě a optimalizaci algoritmu. Pokud jsou zadána „pěkná“ čísla, neprojeví se rozdíly v (ne)přesnostech a grafickou kartu můžeme použít i pro velké matice, kde limit je pouze v maximální velikosti textury grafické karty. Toto omezení velikosti textury lze vyřešit úpravou algoritmu, pokud obětujeme něco z výkonu. Jelikož nepřesnost výpočtu stoupá s velikostí vstupní matice, je potřeba najít kompromis mezi rychlostí, přesností a velikostí vstupních dat.

6. Budoucnost?

Kromě zde probraného DirectX, jsou pro GPGPU výpočty v současnosti používána i další rozhraní, např. OpenGL, Brook, Cg, CUDA. Situace se ale velice rychle vyvíjí, na trh se dostávají nová řešení. V této kapitole upozorním na nejzajímavější z nich. Všechny zmíněné technologie spojuje snaha o větší rychlost, přesnost a univerzálnost.

6.1. MicroSoft DX11

Shader Model 5 představí nový shader nazvaný *compute shader.* Ten je navržen jako pole threadů rozdělených do skupin. Každá skupina má sdílenou paměť spolu s ostatními thready ve stejné skupině, tyto thready pak mohou mezi sebou sdílet např. mezivýsledky. Stejně jako pixel shadery, budou thready moci libovolně číst a nově také zapisovat do grafické paměti s náhodným přístupem. Dále oproti DX10 přibude možnost zapnout dvojitou přesnost, pokud to grafická karta bude podporovat. Maximální velikost textury se zvýší na 16K x 16K. Z programátorského hlediska bude nové HLSL blíže objektovému jazyku.

Idea této technologie vychází z osvědčeného principu CUDA (bude zmíněno dále). MicroSoft konzultuje vývoj DX11 s ATI i nVidia, jeho hardwarová podpora je tedy jistá. Hlavní výhoda, oproti níže uvedeným řešením je hardwarová přenositelnost, nevýhoda závislost na operačním systému.

Podrobnější informace jsou k nalezení v [2], [31], [35], [36].

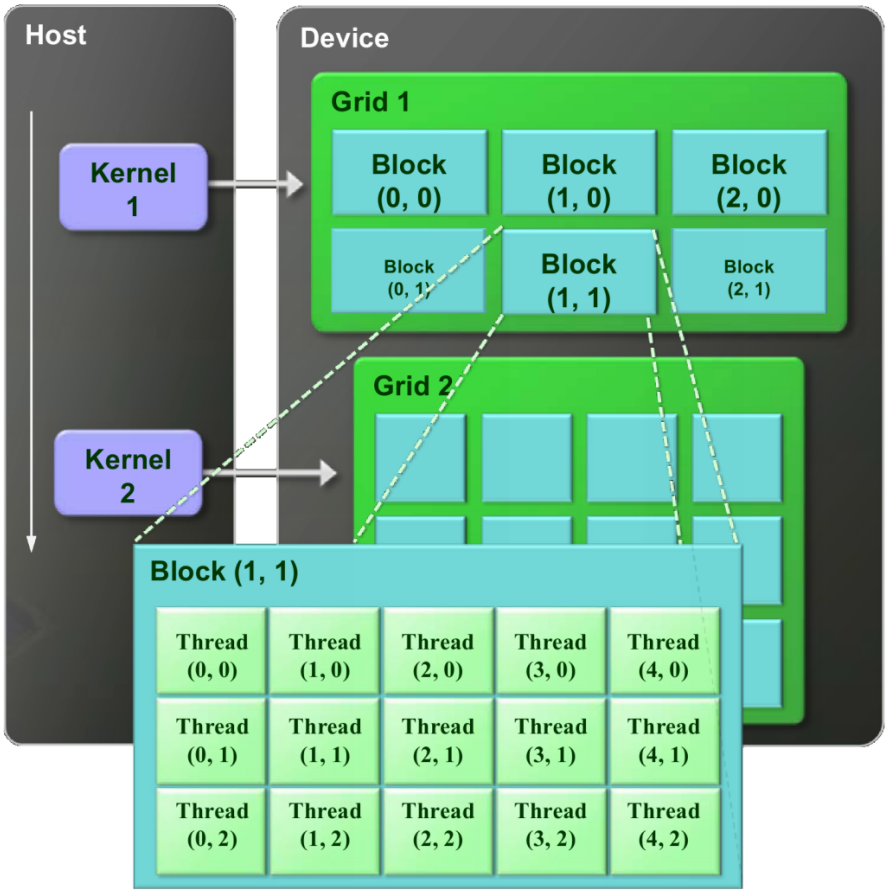
6.2. Khronos OpenCL

Konsorcium Khronos stojící mimo jiné za populárním OpenGL, pracuje na univerzálním rozhraní pro koordinování paralelních výpočtů s názvem OpenCL [10]. To není zaměřeno jen na GPU, ale obecně na všechny procesory. Mezi velké firmy podporující tento standard patří mimo jiné Apple, AMD/ATI [38], Intel, IBM, nVidia. Programovací jazyk pro OpenCL vychází z C99, tedy z C standardu. Mezi hlavní výhody OpenCL patří bezesporu hardwarová a softwarová přenositelnost [37]. Bližší informace k tomuto nadějnému standardu naleznete v [42], [43].

6.3. nVidia CUDA

Paralelní hardwarová architektura popsaná v kapitole 2.6 je doplněna programovacím modelem CUDA [16]. Nad touto architekturou je připravena vrstva programovacích jazyků, jako první podporovaný byl C++. Nyní lze používat i další výpočetní prostředí [39] postavené nad CUDA, např. OpenCL, DX11 Compute, Matlab, Python, Java.

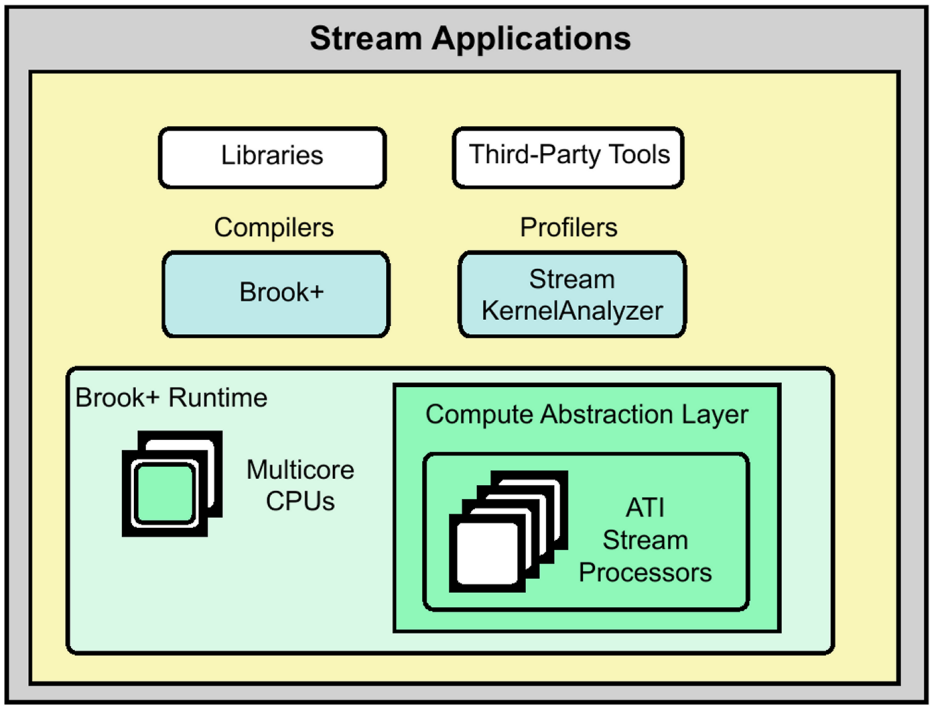
Architektura CUDA je založena na systému threadů, které jsou spojeny do bloků. Ty jsou dále spojovány do sítě bloků (*grid)*. Thready spojené v jednom bloku mohou mezi sebou sdílet data skrz rychlou paměť na čipu. Samozřejmostí je přístup všech threadů do videopaměti s náhodným přístupem.



Obr 18: nVidia CUDA (převzato z [16])

6.4. ATI Stream

Původní rozhraní od ATI se jmenovalo *CTM,* neboli *close to metal* a jak název napovídá, bylo umístěno blízko hardwarové vrstvě. Toto rozhraní nyní ATI vyvíjí pod jménem *CAL (compute abstraction layer)*, blízkost k hardwaru zůstala [27]. Nadstavba nad tento nízkoúrovňový jazyk se nazývá Brook+, který vychází z open source BrookGPU.



Obr 19: ATI Stream (převzato z [27])

Původní BrookGPU je rozšíření jazyka C, funguje i s kartami nVidia, pod OpenGL i DirectX. Verze Brook+ je upravená pro rozhraní CAL, přidává některé nové funkce (např. dvojitou přesnost a scatter) a funguje již jen pod hardwarem od ATI.

6.5. Intel CT - Larrabee

Procesor Larrabee je vysoce paralelní CPU postavený na architektuře x86. Měl by být konkurentem grafických karet v oblasti renderingu i GPGPU se zachováním univerzálnosti CPU. Grafické rozhraní je řešeno čistě na softwarové úrovni. Jeho popis je nad rámec této práce, doporučuji pročíst [8], jde o zajímavou architekturu s nejasnou budoucností.

Intel pro Larrabee vyvíjí vlastní GPGPU jazyk pod zkratkou Ct [41], který bude přirozeným rozšířením C++. Na rozdíl od jazyků od ATI a nVidia je navrhován shora, není tedy stavěn na míru omezujícímu hardwaru.

7. Využití grafických karet pro fyzikální výpočty v počítačových hrách

Do nedávna bylo možné v počítačových hrách simulovat fyziku pouze na CPU. Se zvyšováním realističnosti fyzikálního modelu přestává výkon procesoru dostačovat. Nabízí se možnost použití hardwarové akcelerace podobně, jako v případě 3D grafiky.

Možnosti hardwarové akcelerace fyziky jsou dvě, první z nich je využití specializovaného hardware. Zde se nabízí dedikované řešení od ATI s názvem *FireStream* nebo hardware nVidia *Tesla*. Tyto dvě řešení vznikly upravením technologie současných grafických karet pro potřeby numerických výpočtů a jejich hlavní deviza je (kromě rychlosti) velké množství vlastní paměti. V minulosti do této kategorie spadala karta PhysX od firmy Ageia, na jejím příkladu se ukázalo toto řešení jako nepraktické.

Druhá možnost je ve využití hardwaru, který již většina hráčů doma má. Fyzikální výpočty se dají dobře paralelizovat, grafická karta se tedy nabízí. Hit blízké budoucnosti pravděpodobně bude ve dvou grafických kartách zároveň, jedna se bude starat o grafické a druhá o fyzikální výpočty. Praktické testy využívající toto řešení ukázaly některé neduhy a nekompatibility v ovladačích, vyřešení těchto chyb je ale jen otázka času.

Informace ohledně fyzikálních enginů současnosti byly konzultovány s internetem a to na stránkách [8], [10], [19], [40], [44], [45]. Je třeba si uvědomit, že herní trh je dost veliký a není třeba hledat jednoho absolutního vítěze. Komplexní fyzikální enginy s reálnou možností výpočtů na grafických kartách jsou v momentálně pouze dva:

7.1. nVidia PhysX

V roce 2005 vydala společnost Ageia první kartu určenou výhradně k akceleraci fyzikálních výpočtů. Ageia nabízela potřebný hardware a podporu vývojářům zdarma. Výdělek pak měl přinést prodej specializovaných karet PhysX hráčům. Tyto karty byly vsak příliš drahé a herní základna minimální. Následovala akvizice nVidií, která přepracovala PhysX engine na míru svému CUDA rozhraní. Problém drahého hardwaru byl tedy vyřešen, PhysX je nyní možné počítat na jakékoli GeForce kartě verze 8 a vyšší.

Podobně jako nVidia pohltila Ageiu, tak nejprve Ageia odkoupila dva týmy vyvíjející konkurenční fyzikální engine, NovodeX a Meqon. Do rukou nVidie se tak spolu s kvalitním API dostalo velké množství smluv s herními studii. Licencování enginu

zůstává zdarma, situace pro vývojáře i hráče se tedy značně zlepšila.

Velká nevýhoda je závislost na specifické grafické kartě. Pokud v systému není GeForce, počítá se simulace fyziky na CPU, samozřejmě s určitým propadem výkonu. Není tedy prakticky možné vydat hru stojící na komplikované fyzice, ta musí být stále brána jen jako bonus. Toto omezení snad brzy odpadne, nVidia zvažuje portaci PhysX na OpenCL. Po tomto kroku by bylo možno PhysX akcelerovat i na kartách od ATI nebo například na Larabee. Zda bude na konkurenčním hardwaru stejně výkonná ukáže až čas, s přihlédnutím k hardwarovým možnostem by na kartách ATI ke zpomalení dojít nemělo. V rámci konkurenčního boje je ovšem možné cokoli, např. úmyslná ne optimalizace PhysX na cizím hardwaru. Jestliže se nVidia k přechodu na OpenCL neodhodlá, je zde stále možnost licencování CUDA. Prozatím ATI tvrdí, že CUDA ani PhysX nepotřebuje a spolupracuje s Intelem na implementaci Havoku pod OpenCL. Jedná se pouze o marketingový tah, specifikace CUDA je přístupná a již se ji na ATI kartách podařilo zprovoznit.

Druhou nepříjemností je nedokonalá dokumentace a neúplná implementace některých částí fyziky, engine se stále vyvíjí. Začlenění PhysX do vlastní hry tedy není triviální záležitost ani za podpory nVidie, která ochotně komunikuje. Za zmínku stojí kvalitní debugovací nástroje vycházející z původních řešení Meqonu.

I přes zmíněné nedokonalosti se PhysX rychle šíří, v současné době nemá totiž konkurenci. Relativně silná firma v pozadí navíc znamená, že nedostatky budou pravděpodobně brzy odstraněny. Přestože někteří vytýkají systému jeho uzavřenost, nevypadá budoucnost PhysX vůbec špatně.

7.2. Intel Havok

Historie Havok enginu se píše od roku 2000, kdy vyšlo první SDK, od té doby byl použit ve spoustě herních a filmových titulů. Jedná se o vysoce kvalitní fyzikální engine s přehledným API, který nevyžaduje žádnou speciální kartu - vše je počítáno na CPU. Na rozdíl od PhysX je jeho komerční využití licencováno nemalou částkou, nabízí ovšem některé pokročilejší funkce. Od roku 2005 byla vyvíjena verze Havok FX určená pro výpočet fyziky na grafické kartě, konkrétně byl využíván Shader model 3.0. V té době na vývoji spolupracovala ATI i nVidia. Tento FX systém se bohužel stále nepodařilo dotáhnout do použitelného konce a pravděpodobně byl jeho vývoj zastaven.

Firma Havok, která stojí za stejnojmenným enginem, byla v roce 2007 odkoupena

Intelem. Vývoj však pokračuje dál a to včetně spolupráce s AMD/ATI. Všichni zúčastnění si jsou vědomi, že pro udržení konkurenceschopnosti, je nutné přesunout výpočty na grafickou kartu. Nyní se Havok převádí na OpenCL, hardwarová přenositelnost tedy zůstane. Doposud bylo prezentováno pouze jedno malé demo Havoku nad OpenCL týkající se simulace látek, nVidia si s PhysX drží velký náskok.

Pro vývojáře má Havok tedy dvě nevýhody, jednou z nich je pomalý rozjezd akcelerované fyziky, druhá je cena za licenci. Přesto se hry na tomto systému vyvíjejí a vyvíjet budou, jedná se totiž o léty prověřený engine, v jehož pozadí stojí gigantická firma. Hráčům naopak bude vyhovovat hardwarová nezávislost, důležitým kritériem budou výkonnostní testy, na ty je zatím ještě dost času.

8. Shrnutí na závěr

Při práci na projektu jsem se teprve začal seznamovat s programováním pod Windows API a DirectX (o HLSL nemluvě), výsledný kód tedy zřejmě nebude úplně ideální. MSDN a Google byly hodně nápomocni, přesto donutit shadery spočítat i takto triviální příklad, byl poměrně náročný úkol. To samo o sobě vypovídá o použitelnosti DirectX pro numerické výpočty, jde to, ale není to příjemné. Vývoj však postupuje mílovými kroky, již nyní jsou na světě uživatelsky příjemná rozhraní a do budoucna budou přibývat další. V současnosti se jako nejvýhodnější řešení pro GPGPU jeví CUDA od nVidia nebo Brook+ od ATI.

I přes některé komplikace se zadání práce podařilo splnit, vznikl program *gpgpugs2* pro výpočet soustavy lineárních rovnic metodou Gaussovy eliminace. Na tomto programu byly provedeny testy se zajímavými výsledky. Kompletní okomentované zdrojové kódy pro CPU, DX9 i DX10 naleznete na přiloženém CD, stejně jako použitá vstupní data, naměřené časy a data výstupní.

Dalším praktickým příkladem využití numerických výpočtů na grafických kartách je projekt *folding@home* [10], zabývající se biochemickými experimenty. Simulace těchto experimentů je prováděna na stolních počítačích uživatelů připojených k internetu. Kromě výpočtu na CPU má nyní ATI i nVidia svého *folding@home* klienta. Celkový výkon všech připojených počítačů již překročil 5 PetaFlops, což je téměř 5x více, než má *Roadrunner* [10], nejrychlejší superpočítač současnosti. I v projektu *folding@home* je brán v potaz rozdílný hardware obou výrobců grafických karet a na každou z nich je mířen jiný druh výpočtů.

9. Použitá literatura aneb užitečné internetové odkazy

[1] General-Purpose computation on GPUs

http://www.gpgpu.org

[2] MicroSoft Software Development Kit (nápověda a tutoriály)

http://msdn.microsoft.com/library

[3] reimers.net se spoustou tutoriálů

http://www.riemers.net

[4] extremetech.com s novinkami a rozhovory kolem DirectX 10

http://www.extremetech.com

[5] GPGPU tutoriály Dominika Göddekea

http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu

[6] software development of the computer synthetic hologram with GPU

http://www.geocities.jp/takashi_drive2005/gpuindex.htm

[7] root.cz a jeho seriály nejen o grafických kartách

http://www.root.cz

[8] cdr.cz se zmínkami o GPGPU hardwaru

http://www.cdr.cz

[9] answers.com jako příjemná encyklopedie

http://www.answers.com

[10] wikipedie byla také nápomocna

http://en.wikipedia.org

[11] Gaussova eliminace v High Performance Fortran

http://parallel.ru/docs/Parallel/books/DBPP/text/node90.html

[12] přednášky Mordechai Butrashvily

www.gass-ltd.co.il

[13] C++ tutoriál

http://www.learncpp.com

[14] Bobby Anguelov's blog (DX10 tutoriály)

http://takinginitiative.wordpress.com

[15] DX10 tutoriály a rady na italských stránkách

http://www.notjustcode.it

[16] nVidia GPU programming guide
http://developer.download.nvidia.com
[17] koders.com příklady C++
http://www.koders.com/zeitgeist/cpp
[18] game development (hlavně D3Dbook)
http://wiki.gamedev.net
[19] detailní popis hw grafických karet na extrahardware.cz
http://www.extrahardware.cz
[20] vyčerpávající recenze grafických karet na Slovenském serveru
http://pc.sk
[21] zajímavosti kolem hw grafických karet na stránkách CHIPu
http://chip.cz
[22] tutoriály na codersampler.com (především D3Dbook)
http://www.codesampler.com
[23] matematické přednášky z ČVUT Fakulta strojní
http://www.fsid.cvut.cz
[24] seriál o historii grafických karet od Jiřího Součka
http://pctuning.cz
[25] popis DirectX10 novinek a hardwaru grafických karet
http://www.bit-tech.net
[26] popis GPU na webu computer.org
http://www.computer.org
[27] ATI developer site
http://developer.amd.com
[28] popis architektur 3D karet
http://www.rage3d.com
[29] detailní popis RV770 na
http://techreport.com
[30] článek o RV770 na blogu Jamese Hamiltona
http://perspectives.mvdirona.com/
[31] Tom's hardware
http://www.tomshardware.com

[32] výborný web o hardwaru
http://www.anandtech.com/
[33] GPUs/Data Parallel Accelerators by Dezso Sima
http://nik.bmf.hu/..../GPUs_DP_Accelerators.ppt
[34] Fakulta informatiky Masarykovy univerzity
http://www.fi.muni.cz
[35] novinky kolem DX11 na
http://www.realtimerendering.com
[36] game developer conference
http://gdconf.com
[37] diskuzní fóra na
http://stackoverflow.com
[38] rozhovory s vývojáři uveřejněny na
http://www.guru3d.com
[39] reklamní prezentace
http://hpc.sprinx.cz
[40] technologické novinky
http://fudzilla.com
[41] specifikace Intelu
http://techresearch.intel.com
[42] informace a prezentace nejen o grafických kartách
http://www.siggraph.org/
[43] skupina Khronos
http://www.khronos.org
[44] přehled herních novinek
http://games.tiscali.cz
[45] zprávy z oblasti hardwaru
http://techbit.cz
[46] web zabývající se hardwarem
http://www.behardware.com
[47] numerické metody lineární algebry vyučované na FJFI
http://kfe.fjfi.cvut.cz

10.Dodatek

10.1. použitý software

- MS Visual studio 2005 8.0 – IDE pro C++ a jiné programovací jazyky
  http://msdn.microsoft.com
- FX Composer 2.51 – IDE pro psaní kódu shaderů v HLSL
  http://developer.nvidia.com
- DirectX SDK 9.26 – software pro vývoj DirectX aplikací
  http://www.microsoft.com
- CUDA SDK 1.0 – software pro vývoj nVidia CUDA aplikací
  http://developer.nvidia.com
- PhysX SDK 2.81– software pro implementaci fyziky
  http://developer.nvidia.com
- Stream SDK 1.4 – software pro vývoj ATI Stream
  http://developer.amd.com
- inkscape 0.46 – open source editor vektorové grafiky
  http://www.inkscape.org
- open office 3 – open source textový a tabulkový editor
  http://www.openoffice.org
- paint shop pro 7 – grafický editor
  http://jasc.com

10.2. obsah CD

Přiložený kompaktní disk má následující adresářovou strukturu:

- bin = spustitelná verze programu *gpgpugs2* použitého k testování
- data = použitá vstupní a naměřená výstupní data
- src = zdrojový kód programu *gpgpugs2*
- thesis = tato práce včetně prezentace a zdrojových verzí obrázků a grafů

Kopie tohoto cd je volně ke stažení na adrese http://goldsoft.cz/gpgpugs2

10.3. spuštění programu

Na CD přiložený program *gpgpugs2* slouží k výpočtu soustavy rovnic pomocí Gaussovy eliminace. Je zkompilován ve dvou verzích *gpgpugs2_XP.exe* a *gpgpugs2_VISTA.exe* pro různé verze operačního systému. Podle použitých parametrů je výpočet proveden na těchto zařízeních:

- CPU = na procesoru v jednoduché přesnosti (FP32)
- CPU2 = na procesoru ve dvojité přesnosti (FP64)
- DX9 = na grafické kartě pod DirectX9
- DX10 = na grafické kartě pod DirectX10, pouze pod Windows Vista

Další možné parametry jsou:

- %SIZE% %ITER% = celá čísla, velikost matice a počet iterací
- NONE = výpočet se nebude provádět, slouží k vygenerování vstupních dat
- PRECCOMP = porovnání přesnosti grafické karty s procesorem
- FIXED = jako vstup se použije matice s názvem *input_SIZE.txt* a výstupem je *output_SIZE.txt*, kde SIZE je také zadávaný parametr
- FIXEDOUT = jako vstup je použita matice *input.txt* a výstupem je *output_SIZE.txt*
- /help = zobrazí nápovědu programu

Pokud nebude zadána velikost vstupní matice ani parametr FIXED, jako vstup bude použit soubor *input.txt* a výsledky budou zapsány do *output.txt.*

Nebude-li zadán žádný parametr, použijí se standardně tyto:

DX9 DX10 CPU CPU2 PRECCOMP

Jako vstupní data lze použít náhodná čísla (v tom případě dojde k přegenerování souboru *input.txt*) nebo bude soubor *input.txt* načten přímo. Na prvním řádku tohoto souboru musí být uvedeny rozměry matice oddělené mezerou. Zbytek souboru jsou celočíselné hodnoty matice zadávané intuitivně po řádcích a také odděleny mezerou. Vstup může vypadat například takto:

```
4 3
4 7 7 9
1 6 9 3
4 5 5 7
```

10.4. použité zkratky

ALU = arithmetic logic unit

API = application programming interface

CAL = compute abstraction layer

COM = component object model

CRT = cathode ray tube

CT = C for throughput

CTM = close to metal

CUDA = compute unified device architecture

DX = DirectX

FX = fixed point

FP = floating point

FPU = floating point unit

FLOPS = floating point operation per second

GPU = graphics processing unit

GPGPU = general purpose computing on graphics processing units

HLSL = high level shader language

IEEE = institute of electrical and electronics engineers

PPU = physics processing unit

MS = MicroSoft

MMX = multimedia extension

NaN = not a number

RAMDAC = random access memory digital-to-analog converter

ROP = raster operation processor

RTT = render to texture

SIMD = single instruction multiple data

SIMT = single instruction multiple thread

SPMD = single program multiple data

SP = shader procesor

SPA = scalable processor array

TMDC = transition minimized differential signaling

VLIW = very long instruction word

10.5. minislovníček

anizotropní filtrování = zajišťuje, aby se i vzdálené textury zdály ostré

anti-aliasing = vyhlazení ostrých hran

clipping = ořezání hran mimo výhled

culling = odstranění překrytých hran

encoder = zařízení modifikující informace do požadovaného formátu

engine = hlavní část kódu pohánějící celou aplikaci

gather = operace nepřímého čtení

kernel = kód shaderu

konsorcium = příležitostné sdružení podniků

pixel = **pi**cture **el**ement = nejmenší část informace v obrazu

rasterizace = převedení polygonů na pixely

scatter = operace nepřímého zápisu

texel = **text**ure **el**ement = základní element textury

thread = jedno výpočetní vlákno

transform & light unit = předchůdce dnešního vertex procesoru

warp = skupina paralelních threadů která spouští jednu instrukci