ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta jaderná a fyzikálně inženýrská
Katedra matematiky

GPGPU - provádění vědeckých výpočtů prostřednictvím grafických karet osobních počítačů

Jan Vacata

Zaměření: Inženýrská informatika - Softwarové inženýrství
Akademický rok: 2006/2007
Školitel: Ing. Tomáš Oberhuber

Poděkování

No tomto místě bych chtěl poděkovat především svému školiteli, panu Ing. Tomáši Oberhuberovi, za veškerý čas a energii, kterou mi věnoval. Dodával mi inspiraci a nápady pro další práci a věnoval se mi dokonce i ve chvílích, kdy mne samotného opouštěla motivace. Na jeho přednášce *Paralelní algoritmy a architektury* jsem poprvé zaslechl informaci o „provádění výpočtů na grafických kartách“ a nebýt této přednášky, možná bych se nikdy nebyl ponořil do fascinujícího světa GPGPU.

Také bych chtěl poděkovat všem zkušeným vývojářům a všem dobrým lidem z GPGPU komunity. Nebýt jejich příspěvků a rad z diskuzního fóra na `www.gpgpu.org`, mnoho mých technických problémů by dodnes zůstalo nevyřešeno.

Čestné prohlášení

Prohlašuji, že jsem tuto práci vykonal samostatně a uvedl jsem všechnu použitou literaturu.

Jan Vacata, Praha, 10. září 2007

Obsah

Úvod

Svět numerických výpočtů roste v posledních desetiletích ohromujícím tempem. Různé problémy, které lze převést na úlohy řešitelné numericky na moderních počítačích, se dnes objevují prakticky ve všech odvětvích lidské činnosti. Fyzikální simulace, komplikované matematické problémy, různé formy modelování reality a stovky dalších úloh stravují dnes a denně výpočetní čas počítačů mnoha různých typů a architektur. Složitost těchto architektur přitom roste ruku v ruce se stále se zvyšujícími požadavky na výkon. Zdaleka se přitom nejedná pouze o superpočítače bohatých vědeckých a výzkumných institucí, vždyť dvou a vícejádrové procesory dnes umožňují provádět paralelní výpočty na osobních počítačích každého z nás, počítačové *clustery* jsou k vidění snad na každé univerzitě a zajímavou a výkonnou architekturu *Cell* má doma každý majitel herní konzole Sony PlayStation 3. Autora této práce však zaujala jiná, z uživatelského hlediska velmi dobře známá architektura. Jedná se o grafickou kartu osobních počítačů.

GPGPU [1] je zkratka pro *provádění obecných výpočtů prostřednictvím grafických procesorů*. Jedná se o způsob, jak využít grafickou kartu osobních počítačů jako určitý „koprocesor“ pro provádění rychlých paralelních výpočtů. Cílem této práce je seznámit čtenáře s obecnými principy této techniky a ukázat její použití na jednoduchých příkladech. Důraz bude přitom kladen na teoretické i praktické aspekty programovacího modelu, na popis vlastností této architektury a na možnosti jejich využití. Autor neměl v době psaní této práce k dispozici hardware, na kterém by mohl provádět relevantní měření rychlosti a výkonu, taková měření budou proto předmětem práce navazující.

[1] z angl. General-Purpose computation on Graphics Processing Units

Kapitola 1

GPGPU - obecné principy

1.1 Úvodní motivace

Gordon Moore v roce 1965 předpověděl, že počet tranzistorů, které bude možné umístit na jediný čip, se každý rok zdvojnásobí. Zatímco Intel 4004, považovaný za „pradědečka“ dnešních mikroprocesorů, potřeboval pouze 2300 tranzistorů, dnešní procesory jich obsahují stovky milionů. Spolu s rostoucím počtem tranzistorů zároveň klesá jejich velikost, zvyšuje se rychlost (frekvence) a snižují emise tepla. Uvedené technologické pokroky samozřejmě znamenají stále se zvyšující výkon.

Nejdůležitější vlastností CPU je jejich *univerzálnost*. Většina z miliónů tranzistorů je tak obětována na stále se zvětšující L1 i L2 cache, její řízení, různé metody predikce skoků a toku programu, spouštění instrukcí mimo pořadí a další vlastnosti, které znamenají vyšší výkon CPU jakožto *instrukcemi řízeného* univerzálního výpočetního prostředku. Postavíme-li však CPU před úkol „sečti dva vektory rozměru 10^6“, je nám většina z těchto složitých vlastností k ničemu. Použitý algoritmus bude totiž zřejmě přímočarý, bez větvení a skoků (přesněji s jedním cyklem o známém počtu kroků), přičemž data mohou být zpracována sekvenčně. V takovém případě nás zajímá pouze hrubá výpočetní síla, rychlost přístupu k datům v paměti a schopnost paralelního zpracování. Ukazuje se, že v těchto vlastnostech dnešní CPU samy o sobě příliš nevynikají.

Poněkud stranou pozornosti tvůrců tradičního software se však během posledního desetiletí objevil jiný velmi výkonný hardwarový prostředek. Je jím grafická karta osobních počítačů. Ta kdysi byla jen nutným mezičlánkem pro zobrazení dat, pouhým zásobníkem obrázků. Trh s počítačovými hrami však způsobil, že se z ní stalo druhé výpočetní jádro osobních počítačů. A co je důležitější, grafický procesor (dále jen GPU) dnes v mnoha ohledech překonává klasické CPU. Začněme jednoduchým porovnáním výkonného HW současnosti. Uvažujme:

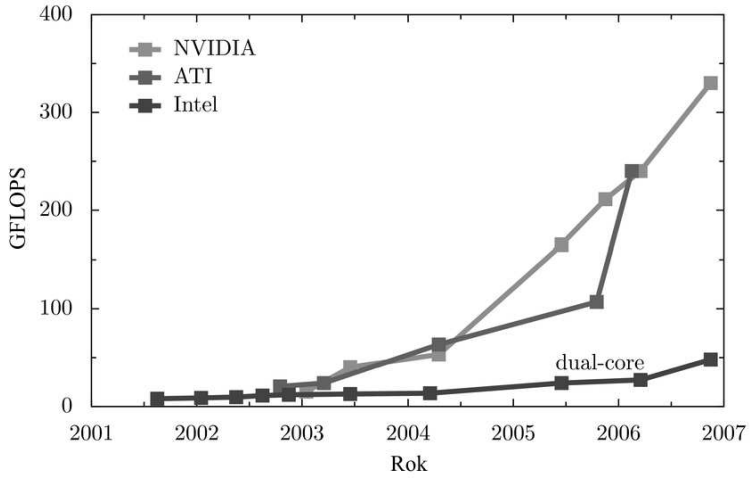
- CPU - Intel Woodcrest Xeon 5160, 3,0 GHz Core 2 Duo (dále jen Intel 5160)
- GPU - NVidia GeForce 8800 GTX (dále jen GeForce 8800)

	Intel 5160	GeForce 8800
Počet tranzistorů	291×10^6	681×10^6
Výkon	48 GFLOPS (peak)	330 GFLOPS (měřený výkon)
Propustnost paměti	21 GB/s	55,2 GB/s
Orientační cena	874 USD	599 USD

Tabulka 1.1: Srovnání základních parametrů CPU vs. GPU

Jak je vidět z předchozí tabulky, výpočetní síla GPU je až neuvěřitelná (vzhledem k ceně). V otázce výkonu nás však zajímá nejen současný stav, ale také vývoj a předpovědi do budoucna.

Nedávná měření ukazují, že výkon CPU se každý rok zvyšuje zhruba s kvocientem 1.41. V hodnocení GPU musíme být o něco obezřetnější, jelikož se (alespoň v prvním přiblížení) jedná o velmi speciální zařízení. Výkon GPU ve zpracování pixelů se každý rok zvyšuje zhruba 1.7krát, ve zpracování vrcholů polygonů je to dokonce 2.3krát. Vývoj posledních let je možné vidět na obrázku 1.1. Zvědavý čtenář se jistě bude ptát, proč GPU v nárůstu výkonu i dalších číslech tak hrubě překonává CPU. Technologický důvod spočívá právě ve specializaci GPU jako takového. Zatímco v případě CPU jsou miliony tranzistorů „obětovány“ jeho univerzálnosti, v případě GPU jsou tyto soutřeďovány pro provádění intenzivních paralelních výpočtů (popisem architektury GPU se budeme zabývat dále). Ekonomický důvod pak vychází z velikosti trhu s počítačovými hrami, nároky na hardware se v tomto směru zvyšují ohromujícím tempem a tomu odpovídají i investice do výzkumu a vývoje grafických procesorů.



Obrázek 1.1: Změny výkonu CPU vs. GPU

Moderní grafická karta je dnes v každém osobním počítači, v předchozích odstavcích jsme vyzdvihli její výkon a výpočetní schopnosti. Z pohledu většiny uživatelů se však jedná o speciální zařízení, určené k hraní počítačových her, v lepším případě ke generování a výpočtům složitých 3D grafických scén. Bylo by však možné využít potenciál GPU k výpočtům, které nemají s počítačovou grafikou vůbec nic společného?

1.2 Porozumění architektuře

V této části stručně připomeneme hlavní architektonické rysy GPU.

1.2.1 Trocha historie

Vývojáři postupně identifikovali čtyři generace grafických procesorů: [1]

1. generace (NVIDIA TNT2, 3dfx Voodoo3) - GPU byla schopna provést rasterizaci trojúhelníků a aplikovat až dvě textury, CPU však musel provádět veškeré transformace geometrie. V této verzi byl nicméně CPU poprvé skutečně „odstíněn“ od aktualizací jednotlivých pixelů.
2. generace (NVIDIA GeForce 256, ATI Radeon 7500) - GPU už uměly provádět transformace geometrie(vrcholů) scény a některé výpočty osvětlení, byla rozšířena množina matematických funkcí pro výpočet výsledné barvy pixelu. GPU této generace bylo možné podrobněji konfigurovat, stále však nebyly v pravém slova smyslu programovatelné.

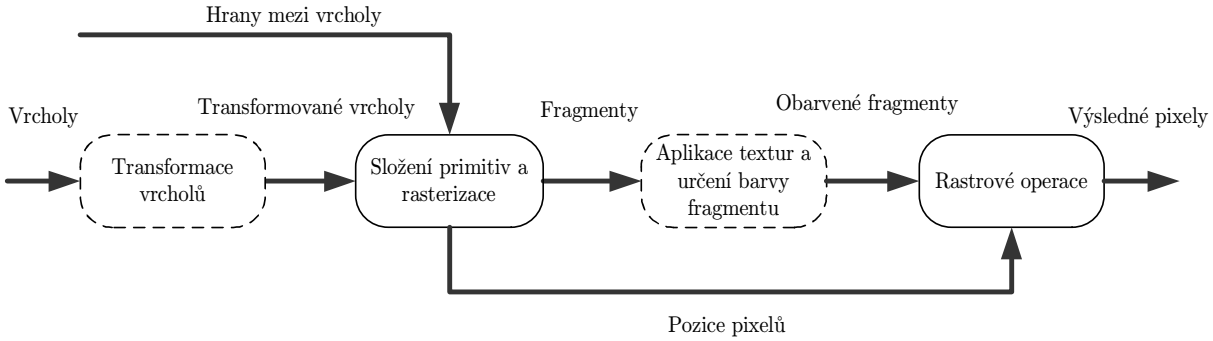
[1] S nástupem jader G80 resp. R600 firem NVIDIA resp. ATI bychom dnes mohli bez mrknutí oka hovořit o páté (skutečně revoluční) generaci GPU, tento pohled si však ponecháme do poslední kapitoly tohoto textu.

3. generace (NVIDIA GeForce 3 a 4, Microsoft XBOX, ATI Radeon 8500) - dá se mluvit o generaci přechodné, poprvé se objevila možnost zpracovávat vrcholy „programátorsky“ pomocí omezené posloupnosti instrukcí. Operace s pixely nicméně programovatelné nebyly, stále se jednalo pouze o konfigurovatelné „zabudované“ funkce (kombinace a filtrování textur, osvětlení, ...)
4. generace (NVIDIA GeForce FX, ATI Radeon 9700) - teprve tato generace GPU umožnila skutečnou programovatelnost na úrovni geometrie vrcholů i výsledných pixelů. Uvedené vlastnosti pak umožnili přesunout většinu zátěže vypočtu 3D scény z CPU na GPU.

Ve všech 4 fázích vývoje GPU se objevovaly snahy o řešení „negrafických“ problémů [3]. Skutečně široce použitelnou je však až poslední uvedená generace GPU, o niž se opírá tato práce.

1.2.2 Grafická pipeline

Na tomto místě nám nezbývá než předpokládat, že čtenář má určité povědomí o 3D počítačové grafice a zobrazování prostorových scén. Všechny současné grafické karty organizují zpracování do tzv. *grafické pipeline*[2]. Její hrubá podoba je uvedena na obrázku. Takové uspořádání se snaží využít přirozené vlastnosti úloh zpracování 3D scén, ty totiž vykazují přirozený paralelismus při požadavku vysoké výpočetní intenzity.



Obrázek 1.2: Grafická pipeline

Tranformace vrcholů Vstupem této fáze je množina vrcholů, z nichž každý si s sebou nese několik atributů (souřadnice vrcholu, souřadnice textury, souřadnice normály, barva, a některé další). Probíhají zde výpočty spojené s vrcholy jako takovými, transformace geometrie a souřadnic textur, výpočty osvětlení (na vrcholech), atp.

Složení primitiv a rasterizace Do této části vstupují tranformované vrcholy spolu s informacemi o spojení mezi nimi. Teprve zde se formují *geometrické objekty* (*primitiva*). Potom dochází k ořezání pohledovým objemem[3] a vyřazení odvrácených ploch[4]. V tu chvíli přichází na řadu rasterizér a interpolátor, které rozdělí primitiva na *fragmenty*[5] a v rámci fragmentů interpolují jednotlivé atributy vstupujících vrcholů.

[2] Autor se záměrně vyhýbá překladům jako *zobrazovací řetězec* a jiné.

[3] z angl. *clipping against view frustrum*

[4] z angl. *back face culling*

[5] Jelikož se v této práci objeví ještě mnohokrát, měli bychom si zřejmě připomenout význam pojmu *fragment* z počítačové grafiky. Fragmentem nazýváme skupinu dat, která je v rámci grafické pipeline použita ke generování (případně aktualizaci) jednoho pixelu ve framebufferu. Každý fragment tak obsahuje interpolované hodnoty jako je barva, normála, souřadnice textury a některé další. Druhou skupinu dat ve fragmentu potom představují ostatní hodnoty, související s rasterizací. Jedná se například o pozici fragmentu v rastru, informaci o hloubce (souřadnice z), atd.

Aplikace textur a určení barvy fragmentu Interpolovaná barva z předchozí fáze zpracování může být zkombinována s jedním nebo několika *texely* (elementy textur). Výstupem této fáze je výsledná barva fragmentu a informace o hloubce.

Rastrové operace V této fázi proběhne několik testů (hloubky, barvy *alfa*, atd.) na vstupujících fragmentech a poté je (v závislosti na zvoleném způsobu *směšování*[6] aktualizován *frame buffer*).

Uvedené řádky představovaly pouze velmi zjednodušený popis základních částí grafické pipeline, více podrobností nalezne čtenář v [1] nebo v [5](spíše softwarový pohled). Z historického pohledu nabízela grafická pipeline pevně dané funkce[7]. Moderní (offline) renderovací systémy však ukázaly, že taková funkcionalita je pro stále se zvyšující požadavky na vizuální kvalitu nedostačující. Ve třetí, resp. čtvrté generaci grafických karet se tak objevila možnost nahradit pevně dané funkce zpracování vrcholů resp. fragmentů uživatelským programem, tak vznikly tzv. *vertex* resp. *fragment programy*. Takové uživatelské programy pak nahradí veškeré funkce dané části původní pevné pipeline (na obrázku 1.2 vyznačeno šrafovaně).

1.2.3 Proudový programovací model

Jak již bylo naznačeno v části 1.1, architektura CPU jako takového jej v mnoha ohledech „brzdí“ v efektivním paralelním zpracování (přestože rozšíření jako SSE nebo 3DNow! umožňují určitou paralelizaci a vektorizaci na úrovni dat). V souvislosti s CPU bychom tak mohli mluvit o tzv. *sériovém programovacím modelu*. Dobrým příkladem tohoto jevu je paměťový systém CPU. Ten je totiž optimalizován na nízkou *latenci* při náhodném přístupu k datům, spíše než na vysokou *propustnost*. Systém několika úrovní cache paměti je poměrně složitý, při sekvenčním jednorázovém přístupu k datům v paměti je však zřejmě k ničemu.

Zcela jinou architekturu i přístup představuje tzv. *proudový programovací model*. V tomto modelu všechna data představují *proud*(*stream*), ten je tvořen uspořádanou množinou dat stejného typu. Přitom elementy proudu mohou tvořit data jednoduchá (např. reálná čísla) stejně jako data složená (např. body, trojúhelníky atp.). Přestože proud může být obecně libovolné délky, skutečně efektivní je práce s velmi dlouhými proudy. Výpočty na proudech provádí *jádra*(*kernels*). Jádro je spouštěno na jednom nebo více proudech a produkuje jeden nebo více proudů výstupních. Jádro typicky aplikuje nějakou funkci na každý element proudu, může také provádět jednu z operací redukce(jeden nebo více elementů je zkombinováno do výstupního elementu), expanze(ve zřejmém smyslu opak redukce), filtrování(výstupem je podmnožina vstupních elementů).

Na tento model jsou kladeny ještě dvě důležité podmínky. Výstupní proudy jádra jsou funkcemi pouze vstupních proudů (ničeho jiného). Výpočet na elementu proudu nesmí záviset na výsledku výpočtu na jekémkoliv jiném elementu. Uvedené vlastnosti dovolují mapovat na první pohled sériové výpočty na proudech na vysoce paralelní zařízení. Požadovaná aplikace je typicky tvořena zřetězením několika takových jader za sebou. Všímavého čtenáře jistě napadne, že grafická pipeline velmi dobře odpovídá tomuto modelu.

Jaké jsou výhody proudového programovacího modelu, jestliže jej nějaký HW implementuje? Model má předpoklady k efektivnímu počítání, protože využívá přirozený vnitřní paralelismus aplikací (takových, které na něj lze namapovat). Popsaná nezávislost mezi elementy proudu umožňuje implementovat datový paralelismus. V rámci zpracování daného elementu je možné využít také instrukční paralelismus. Nakonec zřetězení více jader za sebou může být využito k zřetězenému zpracování (překrývání) jednotlivých jader. Proudový model navíc umožňuje realizovat efektivní komunikaci. Zpracování celých proudů najednou totiž amortizuje cenu za navázání komunikace. Navíc HW implementace dovoluje předávat výstupy jednoho jádra přímo dalšímu jádru v řetězu bez transferu dat „na dlouhé

[6] z angl. *blending*
[7] z angl. *fixed function pipeline*

vzdálenosti“ přes čip. Nakonec kvalitní zřetězené zpracování umožňuje zakrýt latence při práci s pamětí. Celkově lze říci, že celý model je optimalizován na vysokou propustnost (velkých dat) spíše než na latence (náhodných přístupů).

Ačkoliv text uvedený v této části se může zdát příliš „teoretický“, je skutečně dobré uvědomovat si tento model při hledání aplikací dobře mapovatelných na GPU. Pojmy zde uvedené navíc čtenář může najít v mnoha materiálech o GPGPU. Další detaily o proudovém programovacím modelu je možné najít v [7] nebo v [1].

1.2.4 Architektura GPU podrobněji

Jedním z viditelných rysů architektury konkrétních GPU je určitá rouška tajemství, která ji obestírá. K tomuto jevu jistě přispívají jak rychlost, s jakou jsou do GPU implementovány nové funkce, tak poměrně výrazný konkurenční boj mezi dvěma hlavními výrobci grafických karet osobních počítačů. Některé konkrétní principy, údaje a čísla, které by se pro hlubší analýzu možností a výkonu GPU hodily, tak často vůbec není jednoduché získat. Autorovi textu se tak pro zajímavost dosud nepodařilo zjistit velikost cache paměti procesorů jeho grafické karty (přestože na řešení tohoto problému existují experimentální přístupy). Velice dobře zdokumentována (na uvedené poměry) je architektura karet NVIDIA GeForce řady 6(dále jen GF6), podrobné informace je možné nalézt například v [1]. Přestože ta se v době nejmodernějších karet firem NVIDIA i ATI jeví jako zastaralá, pro naše účely zcela postačí a pokryje nutné principy[8]. V následujícím přehledu se budeme soustředit pouze na takové vlastnosti GPU, které mají nějaké praktické využití nebo vazby vzhledem ke GPGPU.

Na výkon každého výpočetního stytému má podstatný vliv rychlost pamětí a přenosová rychlost příslušných sběrnic. Přestože rychlé AGP resp. PCI-E sběrnice umožňují relativně rychlý transport dat směrem CPU$\longleftrightarrow$GPU (pro PCI-E verzi s 32 kanály je teoretické maximum 16GB/s, skutečná rychlost je však typicky mnohem nižší), stále se jedná o určité „úzké hrdlo“ každé aplikace využívající GPU. Větší pozornost si však zaslouží přenosová rychlost mezi GPU a grafickou pamětí. Ty jsou sice z ekonomických důvodů osazovány běžnými paměťmi DRAM, vzhledem ke GPU jsou však tyto uspořádány do několika paralelních bloků, pro GF6 jsou to až čtyři bloky DRAM paměti, to dává šířku sběrnice 256 bitů a teoretickou přenosovou rychlost 35GB/s.

O zpracování a transformace vrcholů se starají tzv. *vertex procesory* (dále jen VP), někdy také označované jako *vertex shadery*. Těch je v HW typicky umístěno několik (až 6 u GF6, až 16 u novějších grafických karet), jsou schopny nezávisle paralelně zpracovávat jednotlivé vrcholy. VP je vektorový procesor, během jednoho taktu je schopen provést čtyřsložkovou MAD operaci[9] a skalární speciální operaci (například instrukce SIN nebo EXP se zřejmým významem). VP může změnit souřadnice libovolného vrcholu a tím ovlivnit pozici výsledných rasterizovaných fragmentů (pixelů). Dnešní VP také mohou přistupovat do paměti textur. Díky tomu můžeme říci, že VP je schopen jak operace *gather*(nepřímé čtení z paměti), tak operace *scatter*(nepřímý zápis do paměti).

Po složení primitiv a rasterizaci převezmou práci tzv. *fragment procesory* (dále jen FP), známé také pod názvem *pixel shadery*[10]. Ty jsou sdruženy do skupin po čtyřech pro rychlé výpočty derivací(diferencí) v texturách. FP jako takové opět nezávisle paralelně zpracovávají jednotlivé fragmenty (v GF6 je až 16 FP procesorů, v nevějších GPU až 48). Opět se jedná o vektorové procesory. Co je

[8] Architektura karet GeForce řady 7 ve skutečnosti nepředstavuje zásadní krok jiným směrem, jedná se spíše o určité zobecnění a rozšíření řady předchozí, takže stále bude odpovídat našemu popisu. Teprve nejmodernější GeForce řady 8 (a podobně ATI RADEON řady R600) představují poměrně velkou revoluci a změnu architektury, která se vymyká našemu popisu. Tento vizionářský pohled si však (jak už bylo uvedeno) ponecháme do poslední kapitoly textu. Později se také krátce budeme věnovat dilematu ATI versus NVIDIA vzhledem ke GPGPU.

[9] z angl. *multiply-add*, známá instrukce pro vynásobení a přičtení

[10] V tomto místě bychom rádi čtenáře upozornili na jistou nekonzistenci v označení, pojem *pixel shader* se v literatuře objevuje ve dvou různých významech, někdy označuje fragment procesor jakožto HW součást, jindy zase vlastní fragment program, spouštěný na FP, stejně dvojsmyslně je používán i pojem *vertex shader*. Popsaná nejednoznačnost však nepředstavuje velký problém, protože význam spojení *pixel shader* je většinou okamžitě zřejmý z kontextu jeho použití.

důležité, všechny FP dohromady se chovají jako SIMD[11] paralelní procesor. To umožňuje zakrýt některé latence při načítání dat z textur, zároveň však přináší komplikace s větvením programů (když je třeba spouštět různé větve programu na různých FP). FP mohou číst data z libovolného místa paměti textur, jsou tedy schopny operace *gather*. Místo v paměti, na který bude proveden zápis výsledku operací FP je však pevně určen (pozicí pixelu ve framebufferu), FP proto v obecném slova smyslu nejsou schopny operace *scatter* (později si ukážeme určité cesty, jak toto omezení obejít).

Shader model 3.0

Vertex i fragment procesory lze považovat i na HW úrovni za nezávislé prováděcí jednotky (PE), mají své množiny instrukcí, vstupní i pracovní registry, podporované formáty čísel, možnosti přístupu do paměti za účelem čtení nebo zápisu a další vlastnosti. Všechny tyto vlastnosti se v posledních letech intenzivně vyvíjely, každá generace grafických karet znamená posunuté dřívější hranice. Přestože existují snahy o standardizaci, každý grafický čip přináší mnoho změn, přičemž podstatné rozdíly samozřejmě existují také mezi GPU jednotlivých výrobců. Ke klasifikaci GPU z tohoto pohledu je výhodné (a vlastně ani jiná možnost není) využít MS DirectX. OpenGL API je totiž vyvíjeno kontinuálně skrze jednotlivá rozšíření (ARB, EXT, atd. viz [8]). DirectX novějších verzí obsahují specifikaci tzv. *shader modelu*. Tabulka 1.2 udává stručný vývoj některých vlastností VP a FP v posledních letech právě s ohledem na implementovaný shader model.

	DX8 SM 1.0	**DX9 SM 2.0**	**DX9 SM 3.0**
Počet instrukcí	128	256	≥ 512
	$4+8$	$32+64$	≥ 512
Konstantní registry	≥ 96	≥ 256	≥ 256
	8	32	224
Pracovní (TEMP) registry	12	12	32
	2	12	32
Vstupní registry	16	16	16
	$4+2$	$8+2$	10
Renderovací cíle	1	4	4
Maximální velikost 2D textury			4096×4096
Počet textur			4
	8	16	16
Řízení toku programu	-	Stat.	Stat./Dyn.
	-	-	Stat./Dyn.

Tabulka 1.2: Vývoj pixel a fragment procesorů. Některé položky jsou rozděleny na dva řádky, vrchní vždy odpovídá vertex procesoru, spodní potom fragment procesoru.

Uvedený přehled vývoje vlastností v žádném případě není kompletní, pro jeho doplnění odkazujeme čtenáře na [11], případně přímo na DirectX SDK. Dále je třeba říci, že shader model představuje pouze specifikaci, kterou jednotlivý HW více či méně dobře implementuje. Shader model 3.0(dále jen SM3) je v současné době stále nejpoužívanější, je implementován naprostou většinou současného HW (včetně GF6), a my jej budeme v dalším textu předpokládat jako nutný základ pro GPGPU vzhledem k této práci (přestože mnoho GPGPU aplikací samozřejmě SM3 nevyžaduje).

Nástup SM3 znamenal určitou unifikaci v přístupu k programování VP a FP. Ty nyní podporují stejné množiny instrukcí, oba typy mohou číst data z paměti textur (u VP s jistými omezeními), oba podporují fp32 (s23e8 reprezentace čísel s plovoucí řádovou čárkou, nejedná se však o plnohodnotnou

[11] Simple Instruction Multiple Data

implementaci specifikace IEEE-754) vnitřní reprezentaci čísel. Oba typy procesorů také podporují dynamické řízení toku programu (větvení a smyčky), i když v případě FP je nutno brát v úvahu poměrně značná omezení (toto bude vysvětleno dále). FP mohou pro výstup použít až čtyři *renderovací cíle* (dále jen MRT[12]) plus informaci i hloubce.

1.3 Základní koncepty

1.3.1 Předpoklady pro GPGPU

V této části shrneme již popsané a uvedeme další vlastnosti GPU, které budeme ve větší či menší míře využívat pro naše GPGPU aplikace. Nebudeme už se zabývat tím, kde a jak jsou jednotlivé vlastnosti v HW implementovány. Nebudeme rozlišovat mezi technologiemi, které jsou implementovány na úrovni HW a těmi, které poskytuje grafické API. V dalším textu i ukázkách kódu budeme (tam kde to bude nutné) důsledně mluvit o OpenGL API, přestože všechny popsané techniky by bylo možné ekvivalentně použít v rámci MS Direct3D API.

Programovatelné procesory
V předchozím textu jsme uvedli dva typy programovatelných procesorů, které se vyskytují v GPU. Počet FP v GPU je typicky mnohem vyšší než počet VP (plyne z mnohem vyšších výpočetních nároků na zpracování fragmentů). Dále je třeba si uvědomit, že FP se vyskytují téměř na samém konci grafické pipeline a výsledek jejich výpočtu je přímo zapisován do framebufferu. Oba tyto fakty dělají z FP užitečnější prostředek pro GPGPU aplikace.

Vnitřní datové typy
V průběhu evoluce GPU se objevily tři datové typy s plovoucí řádovou čárkou, jak už bylo uvedeno, SM3 požaduje pro oba typy procesorů vnitřní reprezentaci čísel ve formátu fp32(s23e8), starší GPU pracovaly s formáty fp24(s16e7) a fp16(s10e5). Pokud to aplikace dovoluje, je doporučeno používat čísla s přesností fp16, protože je lze rychleji přenášet z paměti (připomínáme šířku paměťové sběrnice 256 bitů u GF6). Většina dnešních GPU bohužel neumí pracovat s celočíselnými typy (výjimku tvoří nejnovější GPU podporující DX10), ty musí být vnitřně reprezentovány jako typy s plovoucí řádovou čárkou (čtenář si jistě uvědomuje, jaké to znamená těžkosti a omezení).

Formáty textur
Jak už bylo řečeno, FP umožňují náhodný přístup do paměti textur (gather). Textury tak jsou (bude zdůrazněno v dalším textu) hlavním zdrojem dat GPGPU aplikace. GPU však podporují mnoho různých (téměř nepřehledně mnoho) formátů textur. V každém texelu tak může být „uskladněna“ skalární hodnota (textury typu LUMINANCE), stejně jako čtyřsložkový vektor (RGBA textura). Textura přitom může realizovat jeden z interních formátů, např. fp16, fp32, jeden z formátů čísel s *pevnou* řádovou čárkou a mnoho dalších. Textura typu FLOAT_RGBA32 tak v každém texelu obsahuje čtveřici čísel ve formátu fp32, při rozměru (w, h) tak bude v grafické paměti zabírat $w \times h \times 4 \times 4$ byty. Rozměry textur musely být dříve mocninou dvou, toto omezení však s dnešními GPU a OpenGL rozšířeními neplatí. Aby byla celá situace ještě složitější, různé GPU podporují různé formáty textur, což značně znesnadňuje přenositelnost některých GPGPU aplikací.

Renderování do textury
V tradiční představě grafické pipeline končí výsledek renderování ve framebufferu a je zobrazen na monitoru počítače. Pro GPGPU aplikace však potřebujeme vyřešit následující dva problémy:

[12] z angl. *Multiple Render Targets*

- Načíst výsledek výpočtu zpět do systémové paměti CPU pro další zpracování.
- Pro většinu algoritmů nestačí jeden renderovací průchod, výsledek jednoho kroku potřebujeme podrobit dalšímu výpočtu (většina algoritmů je vždy v nějakém smyslu iterační).

Řešení prvního problému je snadné a spočívá v použití funkcí jako je `glReadPixels` (viz. [5]). Sběrnice mezi GPU a CPU však vždy představuje určité úzké hrdlo, proto bychom takovou funkci měli používat co nejopatrnější (nejlépe realizovat všechny možné výpočty pouze na GPU a až výsledek transportovat zpět do systémové paměti). Tento postup tedy nevyřeší náš druhý problém. V současnosti existují dva možné přístupy. První je často označován jako *kopírování do textury* (CTT[13]) a spočívá v použití funkce `glCopyTexSubImage2D`. Ta nám umožní zkopírovat část (nebo celý) framebufferu do libovolné textury. Mnohem lepší výkon a neduplicitu dat však zajistí metoda *renderování do textury* (RTT[14]), ta využívá OpenGL rozšíření s názvem EXT_framebuffer_object a umožňuje jako výsledek renderování stanovit přímo texturu. Po jednom průběhu renderovacího kroku tak data zůstávají v paměti textur a jsou ihned připravena k dalším výpočtům na GPU. Zde si uveďme jedno důležité omezení. Obecně platí poučka, že jednu texturu nelze použít jako cíl renderovací operace a zároveň z ní číst data. Dokonce i v případech kdy by nedocházelo k principiálním kolizím není výsledné chování ve většině specifikací definováno (a nelze tak na něj spoléhat).

1.3.2 Programovací model

Sebenadšenější čtenář už by si v tomto bodě mohl říci, že bez ohledu na předchozí řádky stále ještě netuší, jak připravit i tu nejjednodušší GPGPU aplikaci. Pokusme se odteď co nejrychleji směřovat právě k tomuto cíli, přičemž budeme ve shodě s literaturou často využívat terminologii z části 1.2.3. Návrh GPGPU aplikace a její implementaci je možné popsat několika kroky.

1. Nejdříve je třeba sestavit hrubý algoritmus, ve kterém dokážeme identifikovat paralelizovatelné části. Každá taková část bude představovat *jádro* a bude implementována jako fragment program. Analogií polí (pro jednoduchost uvažujme nyní pole dvourozměrná), která využívá každý programátor CPU aplikací, jsou v případě GPU *textury*. Ty budou představovat vstup i výstup (proud) každého jádra. Všechny současné GPU samozřejmě podporují *multitexturing*, vstupních proudů tak může být několik, i když toto číslo je omezenou konstantou.
2. Pokud bychom chtěli se vstupním polem dat provést nějakou operaci na CPU, napsali bychom typicky dvě vnořené smyčky, přičemž vlastní jádro by bylo v těle těchto smyček. V případě GPU programátor definuje vhodnou geometrii, namapuje na ní požadovanou texturu (vstupní pole dat) a provede OpenGL volání. Vhodnou geometrii obvykle představuje obdélník rozměrů výstupního pole, orientovaný rovnoběžně s rovinou projekce.
3. Rasterizér grafické pipeline rozdělí tento obdélník na fragmenty, přičemž atributy fragmentů jsou interpolovány na základě jejich hodnot ve vrcholech.
4. Každý takový fragment je potom nezávisle zpracován aktivním jádrem (fragment programem). Na tomto místě je třeba znovu připomenout omezení plynoucí z modelu popsaného v 1.2.3 i z architektury GPU jako takové. Na každý fragment je aplikován *stejný* fragment program. Tyto jeho instance *nemohou* mezi sebou komunikovat a výpočet na jednom fragmentu nemůže přímo ovlivňovat výpočet na fragmentu jiném. Programátor nesmí dokonce uvažovat ani pořadí zpracování jednotlivých fragmentů, toto je zcela v kompetenci GPU. Stále je tedy třeba mít na paměti pojem *nezávislost*. Přestože souřadnice textur jsou v rámci fragmentů také interpolovány, fragment program může tyto souřadnice libovolně aktualizovat a přistupovat tak do libovolného texelu, což je ve shodě s tím co už bylo uvedeno, FP jsou schopny operace *gather*, operace *scatter* však přímo realizovat neumějí.

[13] z angl. *copy-to-texture*
[14] z angl. *render-to-texture*

5. Výstupem každého fragment programu je skalární nebo vektorová hodnota (až 4 RGBA složky) nebo několik takových hodnot (pomocí MRT).

Dosud jsme se nezabývali těžkostmi, které přináší řízení toku programu na GPU. Později si vysvětlíme, že v GPGPU aplikacích nelze tradičně využívat větvení a smyčky (možné to je, ale s mnoha omezeními). Všímavého čtenáře však jistě napadlo, že složitost problémů, řešitelných popsaným způsobem, je řádu $O(n^2)$, což je velmi omezující. Problém libovolné složitosti však dokážeme vyřešit *víceprůchodovým algoritmem*, což je zjednodušeně řečeno několikanásobné opakování uvedeného postupu. Později uvidíme, že takových algoritmů je naprostá většina.

1.3.3 Od teorie k praxi

Snažme se nyní všechny popsané techniky demonstrovat na jednoduchém příkladu, hledejme řešení diskrétní 2D konvoluce metodou GPGPU. Hledáme tedy diskrétní funkci, danou vztahem

$$h_{i,j} = (f * g)_{i,j} = \sum_{k,l=-\infty}^{\infty} f_{k,l} g_{i-k,j-l}$$

kde samozřejmě předpokládáme omezený support funkcí f i g, nechť $|\operatorname{supp} f| = n^2$, $|\operatorname{supp} g| = m^2$, $m \ll n$. Vstupní funkce f i g můžeme tedy reprezentovat maticemi. Je-li m velmi malá konstanta, v naší ukázce zvolíme konvoluční jádro velikosti $m = 3$, jedná se zřejmě o problém řádu $O(n^2)$. Ten je navíc velmi dobře paralelizovatelný a lze jej „nasadit“ na popsaný proudový programovací model (tedy také realizovat metodou GPGPU). Sluší se připomenout, že 2D konvoluce je implementována už na úrovni OpenGL my však popíšeme její implementaci pomocí uživatelského fragment programu.

Každá GPGPU aplikace samozřejmě využívá grafické prostředí daného systému (obecně nezáleží na tom, zda vývojář preferuje MS Windows, X11 na Linuxu, nebo cokoliv jiného, ačkoliv některé diskutované knihovny a nástroje nemusí být nutně portovány na každý operační systém). Nejdříve je tedy potřeba inicializovat nějaké okno, OpenGL kontext, využívaná OpenGL rozšíření atp. Z uvedeného však potřebujeme jen to nejnutnější, naším cílem totiž není vytvářet složité GUI, nebo renderovat komplexní 3D scénu, rádi bychom grafickou kartu co nejjednodušeji donutili „spolupracovat“ a dále už se zabývali výhradně hledáním efektivních GPGPU algoritmů. Navíc bychom ocenili, kdyby naše aplikace byla do určité míry přenositelná mezi operačními systémy (jeden z hlavních argumentů pro upřednostnění OpenGL API před Direct3D API). K tomuto účelu velmi dobře poslouží dvě knihovny, dobře známé OpenGL vývojářům. OpenGL Utility Toolkit (GLUT) je velmi jednoduché API pro psaní OpenGL programů nezávislých na platformě. OpenGL Extension Wrangler Library (GLEW) je knihovna, která usnadňuje práci s rozšířeními OpenGL. Umožňuje snadno zjistit, zda cílový systém podporuje dané rozšíření a v pozitivním případě provede jeho inicializaci. Další podrobnosti o těchto knihovnách nalezne laskavý čtenář ve zdrojích, uvedených v příloze A.

Věnujme se nyní přímo implementaci. Ukázky kódu budeme prezentovat v C++ nebo jakémsi pseudo-C++, doplněném o nutný komentář. Tam kde to bude možné, budeme se snažit hledat jisté analogie mezi klasickým CPU vývojem a tvorbou GPGPU aplikace. Provedení počátečních inicializací, vytvoření okna (které však většinou ani nezobrazujeme - výstup GPGPU programu nemusí mít s grafickými daty vůbec nic společného) a OpenGL kontextu, a inicializaci OpenGL rozšíření zajistí následující úsek kódu:

```
// Inicializace  GLUT a GLEW API, vytvoření okna
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_DEPTH | GLUT_SINGLE | GLUT_RGBA);
//Vzhledem k použití RTT ve skutečnosti není pozice a velikost skutečného okna důležitá
glutInitWindowPosition(0,0);
glutInitWindowSize(width,height);
glutCreateWindow("Convolution filter tutorial");
```

```
glewInit();
```

Dále je třeba nastavit některé atributy OpenGL jakožto stavového stroje. Vstupem renderovací operace je množina polygonů spolu s dalšími informacemi, výstupem potom pole pixelů. OpenGL při renderování transformuje objekty mezi několika souřadnými systémy, které zajistí zobrazení 3D dat do 2D obrazu. My však musíme nastavit tyto transformace tak, abychom zajistili vhodnou korespondenci mezi souřadnicemi vstupní textury, geometrickými souřadnicemi vstupujících polygonů a souřadnicemi výstupních pixelů. Stručně můžeme říci, že potřebujeme zajistit korespondenci 1:1 mezi texely vstupní textury a pixely na výstupu (geometrie a její souřadnice slouží vlastně jako jakýsi „prostředník“ při mapování vstupních texelů na výstupní pixely). Z pohledu grafické pipeline je třeba nastavit rovnoběžné promítání 3D modelu a správnou velikost viewportu. Požadované nastavení zajistí následující funkce:

```
//Mapování 1:1 ve vztahu pixel/texel/geometrie
glMatrixMode(GL_PROJECTION);          //Nastavení rovnoběžného promítání
glLoadIdentity();
gluOrtho2D(0.0, width, 0.0, height);
glMatrixMode(GL_MODELVIEW);           //Potlačení transformace 3D modelu
glLoadIdentity();
glViewport(0, 0, width, height);      //Nastavení viewportu
```

Následuje inicializace vstupních dat. Programátor CPU aplikace by zřejmě inicializoval vhodné 2D pole a do něj namapoval vstupní data. Jak už jsme několikrát zdůraznili výše, analogií polí na GPU jsou textury. Při práci s nimi však musíme být poměrně obezřetní. Základními charakteristikami každé textury je její typ, rozměry, formát a také její interní formát. Typ textury je dán volbou tzv. *texturovacího cíle*. Jelikož hledáme analogii 2D pole na CPU, budeme se v tuto chvíli zajímat výhradně o 2D textury. Stále však ještě máme na výběr ze dvou texturovacích cílů:

- `GL_TEXTURE_2D` je tradiční typ textury, který se nacházel už v nejstarších specifikacích OpenGL. Tento typ textury používá veškeré souřadnice v rozsahu $s \in \langle 0,1\rangle$, $t \in \langle 0,1\rangle$, při práci se souřadnicemi textur je tedy třeba vše přepočítávat do „jednotkové krychle“ (například třetí texel ve třetím řádku textury tak musíme adresovat jako $texel(\frac{3}{W}, \frac{3}{H})$, kde W resp. H je šířka resp. výška textury). V původní specifikaci musely mít navíc textury tohoto typu výšku i šířku rovnu mocnině dvou.
- `GL_TEXTURE_RECTANGLE_ARB` je texturovací cíl, který lze nalézt v nejnovějších specifikacích, nebo v OpenGL rozšíření ARB_texture_rectangle. Tento typ textury nevyžaduje mapování souřadnic do jednotkové krychle ani rozměry rovny mocnině dvou.

O dalších vlastnostech textur jsme se zmínili již v části 1.3.1. Formát ovlivňuje počet hodnot uložených v každém texelu $(1-4)$, interní formát potom způsob, jakým jsou v textuře uložena data[15]. V našem příkladu použijeme jednosložkové textury a následující nastavení (použitelné pro většinu současných GPU firmy NVIDIA):

```
//Texturovací cíl
GLenum texTarget = GL_TEXTURE_RECTANGLE_ARB;
//Formát textury
GLenum texFormat = GL_LUMINANCE;
```

[15] V tomto bodě musíme čtenáře znovu upozornit, že ačkoliv uvedené tři parametry (typ, formát, interní formát) jsou při inicializaci každé textury na GPU naprosto zásadní, je takové nastavení pro začínající GPGPU vývojáře často poměrně obtížné. Důvodem jsou rozdíly mezi jednotlivými výrobci HW, různé grafické karty tak podporují různé interní formáty a problematické jsou často také vzájemé kombinace těchto parametrů. Vnitřních formátů textur existují desítky, výběr vhodného formátu má vliv na rychlost a přesnost výsledné implementace, zároveň však ne všechny formáty podporují metodu renderování do textury tak, jak ji potřebujeme využívat. Více o této problematice lze nalézt ve zdrojích v příloze A.

```
//Vnitřní formát textury
GLenum texInternalFormat = GL_FLOAT_R32_NV;
//Určení formátu dat na straně CPU
GLenum texDataType = GL_UNSIGNED_BYTE;
```

Velikost textury není omezená pouze velikostí paměti GPU (ta je i v dnešní době stále ještě podstatně než systémová pamět CPU), ale také explicitně možnostmi ovladače a firmware jako takového (dnešní GPU typicky podporují 2D textury maximálního rozměru 4096×4096 texelů). Textura zároveň není jenom pole texelů, nese s sebou mnoho informací, které posléze podstatně ovlivňují výsledek jejího mapování. OpenGL (respektive grafický HW) velmi rádo provádí různé interpolace texelů, opakování texturovacího vzorku atp. V GPGPU pojetí však potřebujeme přesnou kontrolu nad hodnotou každého texelu, proto se typicky opíráme o nastavení podobné uvedenému:

```
//Nastavení interpolací nejbližším sousedem
glTexParameteri(texTarget, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(texTarget, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
//Zákaz opakování textur v souřadnicích s i t
glTexParameteri(texTarget, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(texTarget, GL_TEXTURE_WRAP_T, GL_CLAMP);
```

Vlastní inicializaci textury na na straně GPU (v jistém smyslu ekvivalentní volání funkce `malloc` jazyka C při programování CPU) potom zajistí volání funkce:

```
// Inicializace textury na straně GPU
glTexImage2D(texTarget,0,texInternalFormat,width,height,0,texFormat,texDataType, 0);
```

Výše jsme vysvětlili, že „srdcem“ každé GPGPU aplikace jsou uživatelské fragment případně vertex programy. V počátcích programovatelných GPU měli vývojáři k dispozici pouze instrukční sadu využívaných FP a VP a museli je programovat s využitím jazyka nejnižší možné úrovně - assembleru. V dnešní době sáhne vývojář po assembleru pouze v případě, potřebuje-li mít naprostou kontrolu nad kvalitou výsledného kódu, případně potřebuje-li provádět nějaké velmi náročné optimalizace. Pro programování FP a VP je k dispozici hned několik jazyků vyšší úrovně, podrobněji se o nich zmíníme v dalším textu, v našich ukázkách budeme využívat jazyk NVIDIA Cg[16]. Jeho syntaxe je velmi podobná jazyku C a každý vývojář C nebo C++ se v něm snadno zorientuje během chvilky. Bez ohledu na zvolený jazyk pro programování FP musíme napsat zdrojový kód, zkompilovat jej, nahrát na GPU a nakonec jej označit za aktivní kód pro FP, od té chvíle náš kód převezme fázi zpracování fragmentů a bude řídit veškerý výstup této části grafické pipeline (viz obrázek 1.2). Zdrojový kód v jazyce Cg, popisující konvoluci s jádrem rozměru 3×3 je vidět na následujícím výpisu:

```
//NVIDIA Cg fragment program, realizující konvoluci s malým jádrem
struct SInput {
    //Textura mapovaná na renderovanou geometrii, obsahuje vstupní data
    uniform samplerRECT tex;
    //Souřadnice textury, interpolované přes fragmenty
    float2 crd : TEXCOORD0;
    //Uniformní parametr, stejný pro každý zpracovávaný fragment, obsahuje konvoluční jádro
    uniform float[9] conv;
};

//Vlastní tělo fragment programu
float main(SInput inp) : COLOR {
    float outData = 0;
```

[16] z angl. C for graphics

```
    outData +=  (float)texRECT(inp.tex, float2(inp.crd.s-1,inp.crd.t-1)) * inp.conv[8];
    outData +=  (float)texRECT(inp.tex, float2(inp.crd.s,inp.crd.t-1)) * inp.conv[7];
    outData +=  (float)texRECT(inp.tex, float2(inp.crd.s+1,inp.crd.t-1)) * inp.conv[6];
    outData +=  (float)texRECT(inp.tex, float2(inp.crd.s-1,inp.crd.t)) * inp.conv[5];
    outData +=  (float)texRECT(inp.tex, float2(inp.crd.s,inp.crd.t)) * inp.conv[4];
    outData +=  (float)texRECT(inp.tex, float2(inp.crd.s+1,inp.crd.t)) * inp.conv[3];
    outData +=  (float)texRECT(inp.tex, float2(inp.crd.s-1,inp.crd.t+1)) * inp.conv[2];
    outData +=  (float)texRECT(inp.tex, float2(inp.crd.s,inp.crd.t+1)) * inp.conv[1];
    outData +=  (float)texRECT(inp.tex, float2(inp.crd.s+1,inp.crd.t+1)) * inp.conv[0];
    return outData;
}
```

Není účelem této práce být učebnicí nebo snad dokonce referencí jazyka Cg, podrobnosti o syntaxi a možnostech jazyka najde čtenář v [2] nebo ve zdrojích uvedených v příloze A. Nicméně i bez znalosti syntaxe je vidět, že popsaný algoritmus se principiálně neliší od kódu, který by napsal programátor klasické CPU aplikace v jazyce C. Aktivace uživatelského fragment programu na straně GPU spočívá ve volání několika funkcí z NVIDIA Cg API, jejich názvy i s nutným komentářem nalezne čtenář v přiloženém zdrojovém kódu této ukázkové aplikace.

Posledním důležitým krokem konfigurace GPU je určení renderovacího cíle. Výsledkem renderovací operace v klasickém pojetí, je matice pixelů zapsaná do framebufferu (ten je většinou spojen s nějakým oknem na obrazovce). Jak už jsme vysvětlili výše, takové chování nám zejména pro víceprůchodové GPGPU algoritmy podstatně nevyhovuje. Použijeme tedy metodu *renderování do textury*, respektive tzv. *offscreen buffer*, spojený s vhodnou texturou. Přestože metoda RTT je využívána poměrně dlouho (původně bylo využíváno rozšíření WGL_ARB_pbuffer přinášející tzv. pixel buffery), její skutečně jednoduché a hlavně efektivní využití umožnilo až rozšíření EXT_framebuffer_object a jeho *framebuffer objekty* (FBO), přehledný popis tohoto rozšíření nalezne čtenář v [9] nebo přímo v dokumentaci uvedeného rozšíření (lze nalézt ve zdrojích v příloze A). Jeho použití je velice přímočaré. Ke každému FBO je možné připojit několik textur na pozici *bufferu barvy*[17] (odpovídá klasickým framebufferům poskytovaným okenním systémem). Počet takových *přípojným bodů* je však omezen konstantou (u dnešních GPU je tato typicky rovna číslu 4). Z těchto přípojných bodů je potom vždy jeden aktivován a určen jako cíl (renderovací povrch) následující renderovací operace (při víceprůchodových algoritmech potřebujeme často střídat textury, do kterých aktuálně renderujeme). Připojovaná textura musí být v době připojování k FBO nejříve inicializována na GPU uvedenou funkcí `glTexImage2D`. Celý proces vytvoření framebuffer objektu, jeho aktivace, připojení textury a aktivace přípojného bodu je uveden v následujícím výpisu:

```
//Vytvoření FBO jakožto objektu na úrovni OpenGL
glGenFramebuffersEXT(1, &fbo);
//Připojení vytvořeného FBO
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);
//Připojení cílové textury s identifikátorem dstTex na místo přípojného bodu č. 0
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
    GL_COLOR_ATTACHMENT0_EXT, texTarget, dstTex, 0);
//Aktivace tohoto přípojného bodu pro renderování
glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT);
```

Od této chvíle budou všechny renderovací operace nasměrovány do zvolené textury. Uvedené techniky představují základ prakticky každého GPGPU programu. Nyní už máme vše připraveno pro zahájení vlastního renderování. Nastavili jsme vhodné projekce a transformace OpenGL, namapovali vstupní data do textur, ovlivnili jsme chování FP kompilací a aktivací vlastního fragment programu,

[17] Jedná se o neumělý překlad pojmu *color buffer*, známému z OpenGL dokumentace.

jako výsledek renderování jsme určili texturu, v níž nakonec najdeme výsledek jednoho průchodu grafickou pipeline. Pošleme tedy do OpenGL pipeline vhodnou geometrii, v našem případě se zřejmě jedná o obdélník rozměrů supportu funkce f:

```
//Renderuj vhodný obdélník o rozměrech (width, height)
glBegin(GL_QUADS);
    glTexCoord2i(0, 0);
    glVertex2i(0, 0);
    glTexCoord2i(width, 0);
    glVertex2i(width, 0);
    glTexCoord2i(width, height);
    glVertex2i(width, height);
    glTexCoord2i(0, height);
    glVertex2i(0, height);
glEnd();
```

Prakticky všechny GPGPU algoritmy se opírají o renderování podobné geometrie. Další práci přebírá grafická pipeline. Rasterizér vytvoří pole fragmentů, FP na každý z nich aplikují definovaný fragment program. Díky nezávislosti může zpracovávání jednotlivých fragmentů probíhat paralelně a v libovolném pořadí. Rychlost zpracování je navíc přímo úměrná počtu fragment procesorů, implementovaných v HW grafické karty. Víme, že výsledek renderování nalezneme v předem definované textuře. V případě víceprůchodového GPGPU algoritmu bychom tuto texturu namapovali na vstupní geometrii, změnili podmínky (například změnili parametry fragment programu, nebo aktivovali zcela jiný fragment program) a ihned ekvivalentním způsobem zahájili další průchod. V našem triviálním případě jsme však již dospěli k výsledku a nezbývá tedy než načíst jej zpět do hostitelské CPU paměti. Toho lze dosáhnout různými způsoby, například funkce `glGetTexImage` umožňuje načítat pixely z textur. V kombinaci s FBO se však doporučuje použití funkce `glReadBuffer`, která zajistí načtení pixelů z aktivního framebufferu. Přenos výsledku z GPU do systémové paměti je uveden na posledním výpisu:

```
//Přenost výsledku do pole output v systémové paměti CPU
glReadBuffer(fbo);
glReadPixels(0,0,width,height,texFormat,texDataType,output);
```

V uvedených výpisech kódu jsme si odpustili některé drobné detaily, volání ladících funkcí a funkcí pro kontrolu chyb (např. hlášení chyb OpenGL). Jinak ale dostaneme skutečnou GPGPU aplikaci (ačkoliv prozatím velice jednoduchou) prostým spojením uvedených výpisů. Na závěr uveďme ještě stručný komentář.

Shrnutí a závěr

Uvedený fragment program pro konvoluci představuje poměrně naivní a jednoduchou implementaci, která například vůbec nevyužívá instrukční paralelismus. V části 1.2.4jsme si totiž vysvětlili, že FP i VP jsou vektorové procesory, schopny provádět paralelní MAD (na čtyřsložkovém RGBA/XYZW vektoru) během jednoho taktu hodin. Demonstrační příklad však byl vytvořen s důrazem na jednoduchost a přehlednost.

Cílem této části bylo představit čtenáři nejdůležitější softwarové prostředky a techniky programování GPGPU aplikací. V příloze C je možné nalézt úplný výpis zdrojového kódu uvedeného příkladu s mnoha vysvětlujícími komentáři. Autor přiznává, že poměrně velké množství volaných funkcí souvisejících se správným nastavením grafické pipeline, stejně jako orientace ve „filozofii“ fungování grafické pipeline jako takové, může být pro začínající GPGPU vývojáře poměrně obtížné. Pochopení všech principů skutečně zabere nějaký čas a začínající GPU vývojář by jistě měl při čtení této práce nahlížet do referenční příručky OpenGL pro pochopení uvedených funkcí. Na druhou stranu však modifikace jednoduché GPGPU aplikace k řešení složitějších (a tedy i zajímavějších) problémů

už je poměrně přímočará a mnohem méně frustrující záležitost. V posledních letech se navíc objevily knihovny a technologie (o kterých se zmíníme v dalším textu), které se snaží „odstínit“ vývojáře od nutnosti provádět tyto akce ručně a opakovaně v případě každé GPGPU aplikace.

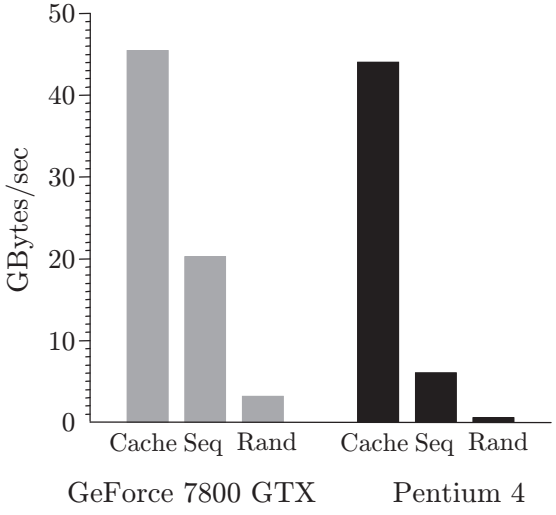
1.4 Hlubší pohled do problematiky GPGPU

Předchozí text nám přiblížil základní principy vývoje GPGPU aplikací. Programovací model musí respektovat možnosti HW a je bohužel velice odlišný od modelu tradičního, používaného vývojáři na CPU. Vždy je třeba mít na paměti, že GPU nejsou univerzálním výpočetním prostředkem v pravém slova smyslu. Jsou vyvíjeny pro efektivní renderování, nikoliv pro obecné počítání. Naším dalším úkolem tedy bude přiblížit si podstatná omezení, která s sebou vývoj aplikací pro GPU přináší, pokusíme se detailněji vyzdvihnout v čem GPU vynikají a popsat rozdíly mezi GPU a CPU. Jedině tak totiž budeme shopni rozlišit, které problémy a algoritmy lze efektivně implementovat prostřednictvím GPU a především nalézt způsob, jakým takovou implementaci provést. Pokud to bude možné a žádoucí, pokusíme se u konkrétních bodů uvést také určitý pohled do budoucna. Některá z uvedených omezení totiž vymizela s nástupem nejmodernějších GPU, jiná by měla být odstraněna ve velmi blízké budoucnosti.

1.4.1 Hledání efektivních algoritmů

Efektivní algoritmy vs. paměť

Je známo, že výkon naprosté většiny CPU algoritmů ovlivňuje *lokalita* dat. Je-li možné veškeré paměťové přístupy provést skrze *cache* paměť, jedná se o ideální algoritmus (přesněji ideální z hlediska přístupu k paměti). Podobný princip platí také pro GPU, avšak s několika podstatnými rozdíly. Obrázek 1.3 zobrazuje výsledky měření propustnosti paměti v případě GPU resp. CPU, byl čten jeden byte fp32 dat nejdříve z cache paměti (opakované čtení ze stejné pozice pole), posléze sekvenčně, nakonec náhodně (toto i další měření je možné nalézt v [12]). Je vidět, že zatímco čtení z cache je



Obrázek 1.3: Srovnání v rychlosti přístupu k paměti mezi GPU a CPU

prakticky stejně rychlé v případě CPU i GPU (ve starších srovnáních dokonce CPU o něco málo vítězilo), v případě sekvenčního čtení z paměti je GPU několikanásobně rychlejší. To není překvapující vzhledem k tomu, že jedním z hlavních úkolů grafické karty je plnit oblasti paměti souvislými daty

textur. Chceme-li tedy dosáhnout optimálního přístupu k paměti, musí GPGPU algoritmus strukturovat data tak, aby používal čtení, blízké sekvenčnímu. Takové chování nám velmi vyhovuje ve snahách o urychlování operací lineární algebry (čtenář nechť si pro jednoduchost představí sčítání velmi dlouhých vektorů).

Další důležitý rozdíl mezi CPU a GPU najdeme právě v roli cache paměti. Je známo, že několikaúrovňová CPU cache se s každou generací procesorů stále zvětšuje. GPU cache je primárně určena k filtrování textur a vzhledem k velikosti takových filtrů je typicky velmi malá. CPU využívá cache při čtení i zápisu, GPU cache je určena pouze pro čtení dat z paměti textur, navíc je optimalizována pro lokalitu ve 2D, právě kvůli filtrování 2D textur (víme, že systémová paměť CPU je chápána sekvenčně, jakoby „jednorozměrně"). Tyto vlastnosti tedy vyžadují podstatně odlišný přístup při návrhu a analýze vhodných algoritmů. V případě CPU favorizujeme algoritmy, které dokáží soustředit výpočty do cache, případně se často snažíme takového chování dosáhnout blokovým uspořádáním. V případě GPU však takový přístup není možný. Podstatné urychlení (přeneseme-li je na GPU) vykazují naopak algoritmy, které vyžadují sekvenční přistup do paměti, jinými slovy takové, které se nesnaží o znovuvyužití už jednou čtených dat (taková data je totiž snadné „nalézt" ve velké CPU cache, v případě GPU to však neplatí).

Aritmetická intenzita

Následující notoricky známá poučka platí stejně tak pro CPU i GPU: dosáhnout maximálního výpočetního výkonu HW znamená mnohem více počítat, než číst data z paměti. Jakékoliv latence při práci s pamětí budou amortizovány, pokud daná aplikace provádí dostatečně mnoho aritmetických výpočtů pro každý načtený byte dat. Chceme-li skutečně využít ohromujícího výpočetního výkonu GPU, musí náš algoritmus vykazovat vysokou *aritmetickou intenzitu*, ta je definována jako:

$$\textit{aritmetická intenzita} = \frac{\textit{počet aritmetických operací}}{\textit{počet bytů transportovaných z paměti}}$$

Z tabulky 1.1 je vidět, že v případě GeForce 8800 musí algoritmus pro dosažení limitního výkonu vykázat aritmetickou intenzitu větší než 23. Toto číslo se bude navíc stále zvětšovat, protože měření ukazují, že roční nárůst výkonu GPU odpovídá zhruba kvocientu 0,7, kdežto nárůst propustnosti paměti odpovídá přibližně kvocientu 0,25. Další měření a výsledky je možné nalézt v [12], případně ve zdrojích v příloze A.

1.4.2 Podstatná omezení

Čísla s plovoucí řádovou čárkou

Soudobé GPU realizují všechny výpočty v aritmetice s plovoucí řádovou čárkou[18]. Nejdříve připomeňme, že každé číslo v této aritmetice je reprezentováno jako

$$\text{znaménko} \times 1, \text{mantisa} \times 2^{(\text{exponent} \ - \ \text{přesnost})}$$

Dvě takové diskrétní aritmetiky se mezi sebou zřejmě liší počtem bitů, přiřazených *mantise* resp. *exponentu*. Ačkoliv dnešní GPU již podporují reprezentaci s23e8 (znaménkový bit, 23 bitů mantisa, 8 bitů exponent), starší modely pracovaly s čísly fp24 ve formátu s16e7 (GPU firmy ATI) nebo fp16 ve formátu s10e5(GPU firmy NVIDIA). Pokud to daný problém dovoluje, doporučuje se použití nižší přesnosti než je fp32, jelikož takový přístup může podstatně zvýšit výkon výsledné aplikace. V souvislosti s tím však musíme mít na paměti problém neexistující celočíselné aritmetiky. To do značné míry zabraňuje hledání GPGPU algoritmů v některých oblastech, typicky například z oboru kryptografie.

[18] Nastupující GPU s podporou DX10 nově implementují také 32b celočíselnou aritmetiku.

Kvůli neexistující reprezentací celých čísel musíme mít na paměti i další důležitý problém, totiž adresování. Tabulka 1.3 uvádí přehled nejvyšších spojitě reprezentovatelných celých čísel v jednotlivých aritmetikách. Ve formátu fp16 tak lze reprezentovat číslo 2 048 a 2 050, ne však už číslo 2 049. Z toho zřejmě plyne velmi důležité omezení při adresování textur. Představme si, že potřebujeme pracovat s velmi dlouhým jednodorměrným polem. To na GPU typicky mapujeme do 2D textury (bude vysvětleno dále). Náš fragment program nyní provede nějaký výpočet jednorozměrné adresy, tu transformuje na 2D adresu textury a načte příslušná data. I kdyby formát fp16 pro vlastní algoritmus znamenal dostatečnou přesnost, při pokusech o adresování vyšších buněk pole než toho s adresou 2 048 začne docházet k přeskakování hodnot, některé elementy pole se nám zkrátka adresovat nepodaří. Taková chyba v implementaci bude navíc velmi těžko odhalitelná.

Formát	**Nejvyšší reprezentovatelné číslo**
fp32	16 777 216
fp24	131 072
fp16	2 048

Tabulka 1.3: Nejvyšší reprezentovatelné číslo, než-li dojde k přeskakování celočíselných hodnot

Vývojáři GPGPU aplikací by velmi rádi v budoucích GPU viděli podporu čísel typu fp64 (jinak řečeno čísel typu `double`). Nemožnost pracovat s čísly ve dvojité přesnosti je v současné době jedním z nepodstatnějších omezení. Nelze však s jistotou předpovědět, jestli a případně kdy se tato možnost v budoucích GPU objeví. Jak už jsme popsali výše, výzkum a vývoj moderních GPU je řízen trhem s počítačovými hrami. Pro výpočet grafických efektů a fyzikálních simulací pro potřeby 3D her totiž fp32 reprezentace čísel zatím zcela postačuje.

Operace *scatter*

Operací *scatter* je míněn nepřímý zápis do paměti na vypočítanou adresu. Čtenář si jistě uvědomuje, že velmi mnoho algoritmů využívá kód tvaru `a[i] = p`, kde `a` je nějaké pole, `i` je vypočítaná adresa a `p` je hodnota, kterou bychom rádi na tuto adresu zapsali. Často navíc potřebujeme provádět oprerace typu `a[i] += p`. Takové konstrukce se vyskytují i v těch nejzákladnějších algoritmech jako je *hashování* nebo *quicksort* třídění. Fragment program nám však žádnou takovou konstrukci nedovoluje, již několikrát bylo zdůrazněno, že výstup fragment programu je pevně dán pozicí fragmentu ve framebufferu a ve fázi zpracování fragmentu už tuto pozici nelze žádným způsobem změnit. Můžeme tedy říci, že FP není schopen operace scatter, přestože bychom tuto velmi často potřebovali. Dále je třeba čtenáře informovat, že nové GPU s podporou NVIDIA CUDA resp. ATI CTM[19] jsou již do jisté míry schopny scattering provádět, ale i pro ně platí jistá omezení:

- není zcela jasné, nakolik použití operace *scatter* ovlivňuje výkon
- je třeba pozorně hlídat řešení kolizí

Ze zmíněných důvodů je vhodné studovat techniky, kterými lze někdy scattering obejít. První možností je přímé převedení operace *scatter* na *gather*. Často citovaným v této technice bývá příklad s fyzikálním systémem hmotných bodů, spojených pružinami (uvažujme jednorozměrný případ). Chceme určit sílu, působící na každý z hmotných bodů. Čtenáře jistě okamžitě napadne algoritmus, který iteruje přes všechny pružiny, pro každou spočítá sílu, jíž daná přužina působí na sousední hmotné body a tuto sílu přičte resp. odečte od těchto „sousedních bodů“. Máme tedy na mysli algoritmus (zapsaný symbolicky):

[19] Zatím nechť si čtenář pod těmito zkratkami představí pouze jakési nové technologie, jejich význam si stručně vysvětlíme v poslední kapitole tohoto textu.

```
pro kazdou pruzinu {
    f = vypocitana_sila();      //Na základě Hookova zákona
    pole_sil_hb[levy] += f;
    pole_sil_hb[pravy] -= f;
}
```

Jak bylo vysvětleno výše, takový algoritmus nelze přímo prostřednictvím GPU implementovat. Snadno si však pomůžeme dvouprůchodovým algoritmem a převedením scatteringu na gathering. Modifikovaný algoritmus by vypadal:

```
pro kazdou pruzinu {            //V prvním průchodu napočítáme síly pružin
    f = vypocitana_sila();      //Na základě Hookova zákona
}
pro kazdy hmotny bod {          //Ve druhém průchodu určíme síly působící na jednotlivé HB
    pole_sil_hb = f[leva_pruzina] - f[prava_pruzina];
}
```

Tento trik jsme mohli využít díky tomu, že spojení mezi hmotnými body je pevně dáno a nemění se v průběhu časových kroků. Jinými slovy pro každou pružinu je od počátku pevně známa adresa, na kterou bychom potřebovali scattering provádět. Tento předpoklad však zřejmě nemusí být obecně splněn. V obecném případě lze využít tzv. *techniku třídění adres*. Předpokládejme že chceme provádět iteraci přes prvky nějakého vstupního pole `in`, pro každý z nich provádět scattering do výstupního pole `out`, pak můžeme použít následující postup:

1. Provedeme iteraci přes všechny elementy pole `in`, na každém napočítáme potřebná data. Na výstup přiřazený elementu pak zapíšeme nejenom tato data, ale také adresu, na kterou bychom rádi zapisovali (kdyby byl dovolen scattering).
2. Vezmeme výstupní pole předchozí operace, setřídíme jej podle adres (například vzestupně).
3. Provedeme iteraci přes prvky výstupního pole `out`, použijeme binární vyhledávání, najdeme všechna data určená k zapsání na tuto adresu a data zapíšeme (kolize řešíme požadovaným způsobem).

Uvedený způsob však vede na mnohaprůchodový algoritmus, časově poměrně náročné bude zejména třídění adres, přestože efektivní algoritmy pro třídění na GPU byly vyvinuty.

Nakonec uvedeme ještě poslední možnost, jak provádět scattering. Ta spočívá ve využití vertex programů. Vertex program totiž ze své podstaty může měnit (a také to často dělá) souřadnice vstupní geometrie. Jak víme, dnešní VP také umožňují přistupovat do paměti textur. Můžeme tedy využít techniku *renderování bodů*. Renderovanou geometrii tak budou tvořit jednotlivé body. Vertex program načte z textur nejen potřebná data, ale také adresy, na které je třeba provádět scattering. Poté změní souřadnice bodů tak, aby odpovídaly získaným adresám. Fragment program pak může dále zpracovat data, výsledek bude nakonec zapsán na požadovanou adresu. Takový přístup však zcela vyřazuje z činnosti rasterizér, práce s jednotlivými body prostřednictvím GPU většinou není příliš rychlá. Typicky se tento postup využívá pouze v případě, potřebujeme-li provádět pouze několik málo operací scatter.

1.4.3 Řízení toku programu

Jedním z prvních programovacích konceptů, se kterými se seznamuje každý začínající vývojář, jsou příkazy pro řízení toku programu, tedy větvení a smyčky. Použití takových konstrukcí se jeví jako fundamentální a nevyhnutelné. Vyvíjet aplikace pro HW, který tyto konstrukce dovoluje jen s podstatnými omezeními, se tak začínajícím GPU programátorům může zdát poněkud frustrující.

Jak víme, GPU je vysoce paralelní zařízení. Můžeme jej brát jako omezenou podobu paralelního stroje se sdílenou pamětí. Některá z těchto omezení jsme již popsali výše (např. nemožnost využít operaci scatter), další si uvedeme právě v této části. V GPU můžeme identifikovat dva typy prováděcích

jednotek, totiž vertex a fragment procesory. Starší typy GPU používaly pro VP i FP výhradně SIMD architekturu a neumožňovaly provádět dynamické větvení programu. S nástupem programovacího modelu SM3 však můžeme v různých částech GPU identifikovat tři způsoby řízení toku programu - *predikaci*, MIMD větvení a SIMD větvení. Jedná se nám zřejmě o řešení konstrukcí typu:

```
if (podm)
    b = f();
else
    b = g();
```

Predikace je nejjednoduší forma větvení. Přesněji se o žádné dynamické větvení nejedná. GPU efektivně projde obě větve smyčky, určí výsledné hodnoty a na základě booleovské podmínky `podm` vybere ze dvou hodnot tu správnou[20]. Větvení metodou predikace podporovaly i starší modely GPU, jeho zjevnou nevýhodou je, že GPU musí vždy vyhodnotit obě větve programu, tento přístup tak bude efektivní jen pro velmi „krátké“ funkce `f` resp. `g`.

Vertex procesory jsou implementovány jako MIMD paralelní stroj a podporují tedy plné dynamické větvení bez omezení (podobným způsobem, jakým jsme zvyklí jej využívat při vývoji na CPU). Naproti tomu fragment procesory realizují architekturu SIMD, v daném taktu hodin je tedy na všech FP spouštěna stejná instrukce. FP také podporují dymanické větvení programu, je však třeba mít na paměti omezení, plynoucí ze samotné podstaty SIMD architektury. FP zpracovávají vždy celý blok fragmentů najednou. Jestliže-je na všech fragmentech v tomto bloku podmínka větvení vyhodnocena stejně, vše je v pořádku a příslušná větev je všemi FP efektivně zpracována. Stačí však, aby jediný FP vyhodnotil tuto podmínku opačně, potom všechny FP musí projít obě větve a teprve správný výsledek je uložen (odpovídá dříve popsané *predikaci*). Jestliže změna ve vyhodnocování podmínky není napříč fragmenty příliš častá (v této souvislosti se často mluví o tzv. *prostorové koherenci*), výkon výsledné aplikace není tímto jevem negativně ovlivněn. Použití SIMD větvení však velmi rychle přestává být výhodné v případech, kdy se výsledek vyhodnocování často mění. Čtenář nechť si představí extrémní případ, kdy podmínka větvení je vyhodnocována na základě náhodně generované hodnoty (se stejnou pravděpodobností hodnoty `TRUE` i `FALSE`), v takovém případě bude použití SIMD větvení trvat stejně dlouho, jako vyhodnocování obou větví programu pro všechny fragmenty. Až dosud jsme navíc nediskutovali režijní náklady odpovídajících instrukcí. Tabulka 1.4 udává, jak dlouho trvá spouštění instrukcí větvení na FP GeForce řady 6. Je vidět, že vytvářet větve kratší než pět instrukcí nemusí být někdy vůbec efektivní.

Instrukce	**Trvání (počet taktů)**
if/endif	4
if/else/endif	6
call	2
ret	2
loop/endloop	4

Tabulka 1.4: Režije instrukcí pro řízení toku programu v architektuře GF6

Z uvedeného by mělo být zřejmé, že instrukce větvení je třeba používat s rozvahou. Z těchto důvodů byly vyvinuty techniky, které přesouvají vyhodnocování větvících podmínek do dřívějších oblastí grafické pipeline (tedy daleko *před* fázi zpracování fragmentů). Většina z nich využívá nějaké pokročilejší vlastnosti grafické pipeline.

[20] Aby nedošlo k mylnému výkladu, zdůrazníme na tomto místě ještě podrobněji rozdíl mezi *větvením* a *porovnáváním*. Dynamickým větvením se myslí použití *instrukcí větvení* resp. *instrukcí skoku*, starší GPU takové instrukce nepodporují. Porovnáváním je myšleno použití instrukce CMP, která odpovídá známému podmíněnému výrazu z C/C++ tvaru `y = podm ? moznost1: moznost2;`. Právě na takové instrukci je založeno větvení metodou predikace.

Statické vyhodnocení
V některých případech zřejmě můžeme výsledek větvení předpovědět ještě ve fázi přípravy dat na CPU. Typickým příkladem je řešení diferenciálních rovnic metodou sítí. Naivní implementace by mohla renderovat vhodnou geometrii tak, aby právě jeden fragment pokrýval jeden bod sítě, fragment program by obsahoval větvení, které by rozhodovalo, jedná-li se o vnitřní nebo hraniční bod sítě. Mnohem efektivnější postup však zřejmě velí renderovat dva druhy geometrie, jeden pokrývající hraniční body sítě, druhý pokrývající body vnitřní. Pak můžeme na každou skupinu bodů použít zvláštní fragment program a tím se vyhnout výše popsanému větvení.

Paměť hloubky
Velmi oblíbená technika předpočítávání využívá *paměť hloubky* grafické pipeline neboli známý *z-buffer*. Zapneme-li v grafické pipeline testování hloubky, pak toto se provádí ještě před vlastním zpracováním fragmentů (tzv. *z-culling*). FP se potom vůbec nezabývají fragmenty, které neprojdou tímto testem, výsledná barva odpovídajících pixelů vůbec není počítána, což jistě šetří mnoho práce GPU. Následující výpis obsahuje kombinaci kódu a pseudo-kódu a naznačuje použití této techniky:

```
//Preprocessing – fragment program vypíše fragmenty na pozicích, které chceme zablokovat
glClearDepth(1.0);
glClear(GL_DEPTH_BUFFER_BIT);
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
aktivuj_fragment_program("preprocess");
renderuj_obdelnik(z=0.0);
//Vlastní aplikační kód
glEnable(GL_DEPTH_TEST);
glDisable(GL_DEPTH_WRITEMASK);        //Zakazujeme další aktualizace z–bufferu
glDepthFunc(GL_LESS);
aktivuj_fragment_program("pozadovana aplikace");
renderuj_obdelnik(z=0.5);
```

Před vlastní aplikaci tedy předřadíme fragment program nazvaný `preprocess`, ten vyhodnotí příslušné podmínky a pomocí příkazu `discard`[21] vypíše pouze takové fragmenty, které mají sloužit k „blokování“. Jelikož tyto fragmenty jsou renderovány se souřadnicí `z=0.0` a všechny fragmenty cílové aplikace budou renderovány se souřadnicí `z=0.5`, ze zpracování budou automaticky vyřazeny všechny fragmenty, které svými souřadnicemi odpovídají nějakému „blokujícímu“ fragmentu. Jsme-li schopni takovéto předpočítání provést, zdá se použití z-cullingu jako ideální technika. Dnešní GPU bohužel implementují z-culling s obecně "hrubším rozlišením", než odpovídá velikosti jednoho fragmentu. GPU tak typicky přeskočí zpracování fragmentu pouze v případě, že nejenom tento fragment samotný, ale také fragmenty v jistém malém okolí neprojdou testem hloubky. Více o této problematice je možné nalézt v [1] případně v [12].

Occlusion query
Poslední technika, kterou si uvedeme v souvislosti s řízením toku programu na GPU využívá tzv. *occlusin query*[22]. Tato součást HW dovoluje přenášet v každém renderovacím průchodu zpět na CPU informaci o tom, kolik pixelů bylo skutečně aktualizováno ve framebufferu. Přenášení těchto (objemem dat nevelkých) informací je navíc implementováno v grafické pipeline tak, aby nijak nesnižovalo její výkon. Představme si situaci, kdy chceme provádět cyklus nějakých renderování až do chvíle, kdy všechny fragmenty splní určitou podmínku. Není nic jednoduššího než, do těla cyklu vložit navíc fragment program (a s ním navíc jeden renderovací krok), který na fragmentech testuje požadovanou

[21] implementován v NVIDIA Cg
[22] Autor se po úvaze záměrně vyhýbá překladům tohoto pojmu.

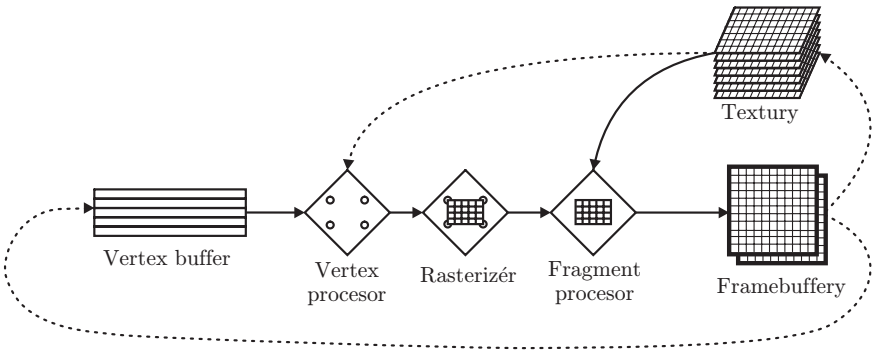
podmínku a v případě jejího splnění vyřadí daný fragment s použitím `discard`. Splní-li všechny fragmenty uvedenou podmínku, je také všem zabráněno aktualizovat framebuffer a occlusion query vrací hodnotu 0, kterou přečte CPU a ukončí uvažovaný cyklus.

1.5 Datové struktury

Jedním ze základních konceptů abstrakce, používaném prakticky na všech úrovních vývoje SW, je formulace daného problému v řeči datových struktur a operací na těchto strukturách. Vývoj datových struktur v rámci programovacího modelu CPU je velice přímočarý, pomocí pointerů můžeme přistupovat na libovolné místo v paměti, překladač za nás obstará adresaci elementů libovolných polí, fantazii vývojáře nejsou do cesty kladeny prakticky žádné překážky (ty se objeví teprve ve chvíli, kdy začneme studovat efektivitu implementace konkrétních datových struktur). V této části se pokusíme popsat nejdůležitější omezení, související s vývojem efektivních datových struktur na GPU. Jelikož v této i navazujících pracích nás bude nejvíce zajímat GPGPU implementace některých operací lineární algebry a numerické řešení diferenciálních rovnic (ODE, PDE, ...), budeme se soustředit především na související datové struktury jako jsou hustá a řídká pole resp. matice.

1.5.1 Paměťový model

GPU podobně jako většina ostatních mikroprocesorů mají svou vlastní hierarchii paměti, na úrovni HW tak lze identifikovat DRAM pamět, cache paměť, dočasné a konstantní registry atp. Tato hierarchie však byla vytvořena pro urychlení grafických výpočtů a odpovídá popsanému proudovém programovacímu modelu. Paměť GPU tak podléhá celé řadě omezení, grafické API jako je OpenGL nebo DirectX navíc dovoluje k ní přistupovat pouze skrze grafické abstraktní pojmy.



Obrázek 1.4: Grafická pipeline v rámci proudového programovacího modelu

Ve shodě s proudovým modelem můžeme v grafické pipeline identifikovat několik základních datových proudů. Obrázek 1.4 zachycuje tyto proudy tak, jak jsou implementovány v GPU podporujících DX9[23]. Vertex procesory tak zpracovávají *proud vrcholů*, fragment procesory potom *proud fragmentů*, dále můžeme mluvit o *proudu textur* a *proudu framebufferu*. Proud textur jako jediný umožňuje náhodný přístup k jednotlivým elementům (texelům), textury mohou být deklarovány a posléze adresovány jako 1D, 2D nebo 3D struktury. Již víme, že technika renderování do textury umožňuje přesměřovat proud framebufferu přímo do textury, nejnovější rozšíření podobně dovolují přesměřovat tento proud do vertex bufferu[24]. MRT dovoluje použít více renderovacích cílů (v dneš-

[23] Grafické karty podporující DirectX 10 vnášejí do grafické pipeline několik nových prvků.

[24] Výstup jednoho průchodu grafickou pipeline tak může být použit jako zdroj geometrie pro průchod následující, tato technika je v literatuře často označována jako *render-to-vertex-array*, případně *render-to-vertex-buffer*.

ních GPU tak každý FP může v jediném kroku zapsat až 16 fp32 hodnot). Vertex/fragment programy (jádra) čtou/zapisují odpovídající proudy (zápis je prováděn pouze na konci jádra), výstupní adresa však je pevně určena pozicí elementu v proudu a jádra tak neumožňují scattering (víme, že VP mohou díky geometrickým transformacím vrcholů „zprostředkovaně“ provádět scattering do paměti textur, implicitní scattering do libovolného elementu proudu vrcholů však samozřejmě také provádět nemohou, tím by byl totiž narušen paralelismus zpracování proudů). Uvedený model vystihuje vnitřní strukturu GPU, kromě ní však musíme uvažovat také okolní paměťové prostředky a řídící schopnosti hostitelského systému, voláním funkcí grafického API se tak CPU stará o alokaci a uvolňování paměti na GPU, kopírování dat mezi CPU $\longleftrightarrow$ GPU, připojování textur pro čtení, připojování framebufferů pro zápis atd.

1.5.2 Efektivní datové struktury

Uvedené vlastnosti GPU je třeba mít na paměti při návrhu a hledání efektivních datových struktur. Jelikož proud textur jako jediný umožňuje náhodný přístup (gathering) a paměť textur zároveň poskytuje dostatečnou kapacitu, používají se právě textury k uložení naprosté většiny dat na GPU. Nejpoužívanějším kontejnerem pro fyzické uložení dat jsou přitom 2D textury. Ty totiž přirozeně odpovídají způsobu, jakým GPU zpracovává data - rasterizér pracuje ve 2D, framebuffer představuje 2D objekt.[25] Další důvod spočívá v omezení velikosti textur, dimenze textury je omezena konstantou (viz. tabulka 1.2). Reprezentace 1D pole pomocí 1D textury by tedy značně omezovala jeho velikost.

Kromě fyzického uložení dat se musíme také zajímat o to, jakým způsobem budeme vlastně k jednotlivým elementům datových struktur přistupovat. Programátorům C++ (i jiných jazyků) je jistě dobře znám pojem *iterátorů*. Můžeme říci, že většina moderních algoritmů pracuje na principu iterace elementů datových struktur. V rámci proudového programovacího modelu nás budou nejvíce zajímat *proudové iterátory*, které postupně procházejí jednotlivé elementy proudu a umožní jeho paralelní zpracování. GPU provádí implicitní iteraci přes elementy datových proudů, popsaných v části 1.5.1. Naším cílem je potom „namapovat“ na tyto proudy požadované iterační přístupy k vlastním datovým strukturám[26].

Hustá pole libovolné dimenze

Pole je jistě nejzákladnější datová struktura. Její nejdůležitější součástí je *překladač adres* (při programování CPU s využitím jazyků vyšší úrovně za nás tuto činnost typicky obstará překladač). Předpokládejme pole dimenze nD. Zatímco v případě CPU překládáme tyto adresy na 1D adresy systémové paměti, v případě GPU volíme z popsaných důvodů překlad na 2D adresy a fyzické umístění dat v texelech 2D textur. Často je navíc možné využít rasterizér nebo VP a provést předpočítání takového překladu adres ještě před vlatní fází zpracování fragmentů, více o takových optimalizacích lze nalézt v [14] nebo v [1]. Nejpřirozenější datovou strukturou na GPU je tedy 2D pole respektive hustá matice. Takové pole lze adresovat přímo, přístup k okolí libovolného prvku navíc probíhá v souladu s organizací cache paměti GPU (do ní jsou data načítána ve 2D blocích). Poslední poznámka k hustým polím se bude týkat iterace přes jejich elementy. Tu zřejmě obstará renderování vhodných obdélníků, pokrývajících požadované oblasti.

Řídká pole, řídké matice

Schopnost efektivně reprezentovat řídké matice je základním předpokladem pro řešení mnoha numerických problémů. Zatímco práce s hustými poli na GPU je veskrze přímočará, řídké datové struktury

[25] GPU s podporou DX10 umožňují renderovat do řezu 3D textury.

[26] Poznamenejme ještě, že v těchto úvahách představuje *iterátor* pouze abstraktní pojem přístupu k elementům datových struktur.

vyžadují mnohem citlivější přístup. Na tomto místě je třeba znovu připomenout dvě nejzákladnější omezení:

1. Aktualizace řídkých datových struktur často zahrnuje potřebu provádět zápis na vypočítanou adresu, FP však neumožňují přímo provádět scattering.
2. SIMD architektura FP značně ztěžuje iteraci přes jednotlivé elementy řídkých struktur.

Poměrně dobře řešitelná je implementace *statických řídkých matic*. Jejich struktura (struktura nenulových prvků) je pevně dána a nemění se během výpočtů. Toho lze využít k relativně snadnému mechanizmu uložení dat a překladu adres. Jeden z jednoduchých způsobů zahrnuje uložení řídké matice řádek po řádku do jediné textury (přesněji ukládají se jen nenulové prvky) ve formě tzv. *segmentů*. Ve dvou dalších texturách jsou pak uchovány počátky segmentů resp. sloupcové indexy. Překlad adres pak probíhá na základě dvouúrovňového *závislého čtení* z paměti textur (závislým čtením rozumějme případ, kdy v texelu textury je uložena adresní informace pro přístup do textury jiné, o takovém texelu můžeme uvažovat jako o pointeru). Kvůli omezením SIMD architektury musíme poté renderovat geometrii přes segmenty stejné délky (tedy přes řádky matice o stejném počtu nenulových prvků). Více detailů o tomto způsobu reprezentace řídkých matic je možné se dočíst v [13]. Zcela odlišný přístup k reprezentaci řídkých matic je možné nalézt v [14] resp. v [1]. Každý nenulový prvek matice je uložen jeko jeden geometrický vrchol. Iterace přes prvky matice pak probíhá renderováním těchto vrcholů (o rozměru jednoho pixelu). Efektivní využití této techniky umožňují tzv. *vertex-buffer objekty*, implementované v novějších verzích OpenGL[27]. Tato funkce dovoluje hostitelské aplikaci specifikovat pole vrcholů a uložit jej přímo v paměti GPU, vlastní renderování velkého množství vrcholů je potom zřejmě podstatně rychlejší.

Implementace dynamických řídkých polí na GPU je podstatně složitější. Mohou-li nenulové prvky náhodně vznikat a zanikat, potom nemožnost provádět scattering znamená velmi výrazné omezení. Nicméně i na tomto poli bylo dosaženo výrazných úspěchů, velmi efektivní implementace dynamických řídkých polí se opírají o tzv. *stránkování* - jedná se o mechanizmus známý z moderních architektur a operačních systémů (čtenář jistě slyšel o pojmech *virtuální paměť* resp. *stránkování paměti*). Virtuální paměť (například 3D řídké pole) i fyzická paměť (typicky 2D textura) jsou rozděleny na tzv. *stránky* stejné velikosti (bloky dat). Stránkovací tabulka potom mapuje využívanou podmnožinu všech možných stránek virtuální paměti na stránky paměti fyzické. O alokaci a uvolňování paměti se stará tzv. *správce stránek*. Překlad adres i management stránek však už je v tomto případě poměrně složitý, efektivní implementace vyžadují jistou pravidelnou komunikaci mezi GPU a CPU, více o této technice je možné nalézt v [15].

1.6 Vývojové prostředky

Vývoj moderních aplikací pro libovolnou cílovou architekturu vyžaduje alespoň několik základních nástrojů. Snad žádný programátor si svou práci nedovede představit bez programovacích jazyků vyšší úrovně, nástrojů pro ladění chyb a analýzu kódu. V poslední části této kapitoly se tak budeme krátce věnovat jazykům, knihovnám a softwarovým prostředkům, které lze s úspěchem použít pro vývoj GPGPU aplikací.

Jak bylo řečeno v předchozím textu, s prvními generacemi programovatelných GPU bylo třeba využívat assembler, nebo přímo specifické jazyky jednotlivých výrobců (viz. např. OpenGL rozšíření ARB_vertex_program a ARB_fragment_program). Přestože v určitých případech může mít jejich použití stále smysl (např. potřebuje-li vývojář mít maximální kontrolu nad efektivitou výsledného kódu spouštěného na GPU), dává dnes naprostá většina vývojářů přednost jazykům vyšší úrovně. Takové jazyky existují celkem tři, velmi dobře vystihují použitý HW a jsou velmi podobné jazyku C. HLSL[28] je vyvíjen firmou Microsoft jako součást Direct3D API, bez něhož není samostatně použitelný. Filo-

[27] Dříve se jednalo o rozšíření ARB_vertex_buffer_object

[28] z angl. High Level Shading Language

zofií velmi podobný je jazyk Cg z dílny NVIDIE, jeho největší výhodou je přenositelnost - programy napsané v Cg lze kompilovat a používat v rámci Direct3D i OpenGL. GLSL[29] byl vyvinut konsorciem OpenGL ARB jako součást OpenGL API, původní specifikace byla v určitém smyslu „vizionářská“ a obsahovala konstrukce, které v té době nebylo možné mapovat na existující hardware (GLSL tak například rozeznával celočíselný typ `integer`, ačkoliv ten jak víme nebyl do nástupu DX10 implementován v HW). Ve stručnosti se dá říci, že nezáleží na použitém jazyku pro psaní shaderů, všechny poskytují v podstatě ty samé funkce a všechny jsou stejně dobře využitelné pro GPGPU. Všechny tyto jazyky byly vyvinuty jako nástroje pro úpravy geometrie a stínování polygonů. Nelze je tedy využít bez znalosti grafické pipeline a grafického API, všechny intenzivně využívají grafických pojmů jako je *vrchol*, *fragment*, atp. Právě jejich původní určení tak představuje největší slabinu z pohledu GPGPU. V části 1.3.3 jsme si naznačili, jak mnoho kódu je potřeba napsat pro správnou konfiguraci grafické pipeline a formulaci požadovaného problému v řeči geometrie, textur, framebufferů atd. Přitom většina z těchto konstrukcí se opakuje ve všech GPGPU aplikacích, činí je nepřehlednými a obtížně pochopitelnými pro začínající vývojáře, přičemž většina GPGPU aplikací nemá s grafikou nic společného.

Z popsaných úvah vykrystalizovalo v průběhu uplynulých let několik knihoven a programovacích nástrojů ještě vyšší úrovně, které se více či méně úspěšně snaží odstínit programátory od grafických pojmů a umožnit jim soustředit se na algoritmizaci vlastních problémů. Tyto nástroje přinášejí vyšší úroveň abstrakce a v jistém smyslu umožňují „klasické“ programování tak, jak jej známe na CPU - formulaci datových struktur, operátorů a práci s pamětí. Je však třeba zdůraznit, že se jedná pouze o abstrakci, která přináší větší pohodlí vývojářům. Vždy je třeba mít na paměti omezení, vyplývající ze samotné architektury GPU, z nichž mnohá jsme si uvedli v předchozím textu. Žádný z uvedeným systémů nám tato omezení neumožní obejít. Některé z těchto systémů nám tak například umožní provádět scattering, ale taková operace *nebude mapována na GPU* (výpočet proběhne normálně na CPU), tudíž nebude efektivní. Vyčerpávající přehled všech těchto knihoven a systémů nalezne čtenář ve skvělém GPGPU přehledu [3] (jemuž tato práce v žádném případě nemůže a nechce konkurovat). Krátce se zmíníme alespoň o těch nejznámějších.

Brook [16] byl vyvinut jako programovací systém pro GPGPU, jedná se o určité rozšíření jazyka C o datově-paralelní konstrukce. Programovací model systému Brook se velmi blíží proudovému programovacímu modelu, popsanému v části 1.2.3. Vstupní i výstupní data je tak potřeba namapovat do *proudů*, vlastní operace na jádrech probíhají paralelně voláním zvláštních funkcí - *jader*, interpolátor GPU je využíván skrze zvláštní *iterátorové proudy*, atd. Proudy v systému Brook jsou velmi podobné klasickým polím, mají svůj datový typ, rozměr a tvar (počet dimenzí), Brook sám se potom stará o jejich mapování do textur na GPU a další související činnosti. Podívejme se na uvedený příklad konvoluce, její jednoduchá (do jisté míry naivní) implementace v systému Brook by vypadala následovně. Vlastní konvoluci realizuje jádro z následujícího výpisu:

```
//Funkce typu jádro systému BROOK
kernel void conv(float inStream[][], out float outStream<>, iter float2 tr<>,
    float3 cmatrixR1, float3 cmatrixR2, float3 cmatrixR3) {

    float2 t0 = float2(-1.0f,-1.0f);
    float2 t1 = float2(0.0f,-1.0f);
    float2 t2 = float2(1.0f,-1.0f);
    float2 t3 = float2(-1.0f,0.0f);
    float2 t4 = float2(0.0f,0.0f);
    float2 t5 = float2(1.0f,0.0f);
    float2 t6 = float2(-1.0f,1.0f);
    float2 t7 = float2(0.0f,1.0f);
```

[29] z angl. OpenGL Shading Language

```
    float2 t8 = float2(1.0f,1.0f);

    outStream = 0;
    outStream += inStream[tr+t0]*cmatrixR3.z;
    outStream += inStream[tr+t1]*cmatrixR3.y;
    outStream += inStream[tr+t2]*cmatrixR3.x;
    outStream += inStream[tr+t3]*cmatrixR2.z;
    outStream += inStream[tr+t4]*cmatrixR2.y;
    outStream += inStream[tr+t5]*cmatrixR2.x;
    outStream += inStream[tr+t6]*cmatrixR1.z;
    outStream += inStream[tr+t7]*cmatrixR1.y;
    outStream += inStream[tr+t8]*cmatrixR1.x;
    outStream = max(0,outStream);
    outStream = min(255,outStream);
}
```

kde `inStream` resp. `outStream` je vstupní resp. výstupní proud dat (funkční hodnoty pro konvoluci), `tr` je iterátorový proud, který po spuštění na GPU nebude představovat nic jiného než interpolované souřadnice textury, `cmatrixRi` jsou konstantní parametry (budou uloženy v konstantních registrech GPU) v nichž je uložena konvoluční matice. Namapování dat do proudů a spuštění paralelního výpočtu potom zajistí několik málo řádků kódu v následujícím výpisu:

```
//Definice proudů a volání jádra
float3 cmr1 = float3(0, -1, 0);
float3 cmr2 = float3(-1,5,-1);
float3 cmr3 = float3(0,-1,0);

float sin<width, height>;
float sout<width, height>;
iter float2 itcoord<width, height> = iter(float2(0.0f,0.0f), float2(width, height));

streamRead(sin, input);
conv(sin,sout,itcoord,cmr1,cmr2,cmr3);
streamWrite(sout, output);
```

Přestože čtenáři nemusí být na první pohled jasný význam některých klíčových slov jazyka Brook, princip by měl být zřejmý. Je vidět, že v tomto případě jsme provedení konvoluce na GPU dosáhli s použitím výrazně menšího množství kódu nežli v části 1.3.3. Veškerá nastavení grafické pipeline stejně jako komunikaci s runtime knihovnou Cg (Brook využívá jazyk Cg, jádra jsou překládána nejprve do jazyka Cg, pak teprve dojde k překladu do instrukcí a kompilaci na GPU) za nás obstará systém Brook. Ten se skládá z překladače a běhového prostředí, které dokáže výsledný kód kompilovat a spouštět pod různými cílovými platformami. Brook aplikace tak jsou přenositelné, je možné je spouštět na GPU v rámci OpenGL nebo Direct3D, stejně jako na CPU (užitečné pro ladění).

Brook však není jediným jazykem vyšší úrovně použitelným pro GPU, Sh [17] je metaprogramovací jazyk pro psaní shaderů (připomeňme že *shaderem* nazýváme program spouštěný na VP resp. FP, určený pro transformace geometrie resp. stínování polygonů) postavený nad C++ (se všemi jeho „silnými zbraněmi“ jako jsou objekty, šablony atd.), jedná se o implementaci tzv. *algebry shaderů*. Stručně řečeno se jedná o myšlenku vytvářet složitější programové celky kombinací a spojováním malých elementárních subprogramů. Tento systém byl vytvořen jako alternativní nástroj pro psaní shaderů (nikoliv přímo pro GPGPU), má v sobě však implementované objekty jako je *proud* (dat) a je tedy pro GPGPU také velmi dobře použitelný. Vývoj tohoto open-source systému však byl zastaven a jeho autoři přešli na vývoj komerčního produktu s názvem RapidMind Development Platform.

Odlišnou cestou se vydali autoři knihovny Glift [18]. Uvědomili si, že návrh a implementace datových struktur na GPU jsou pro vývojáře poměrně frustrující. Tam, kde programátor na CPU přes operátor indexování snadno přistupuje k polím libovolných (často složených) datových typů, musí programátor na GPU ručně řešit překlad adres v rámci jednotlivých shaderů. Moderní pojetí vývoje aplikací se navíc snaží oddělovat návrh datových struktur od vlastní implementace logiky algoritmů. Glift je šablonová knihovna myšlenkově do určité míry podobná knihovnám jako je STL nebo Boost. Umožňuje vývojářům navrhovat znovuvyužitelné datové struktury, paralelně přistupovat k jednotlivým elementům složitějších struktur pomocí *iterátorů*, to vše s minimálním množstvím ručně psaného kódu. Ačkoliv uvedený návrh abstrakce je jistě zajímavý, autorovi této práce se nepodařilo nalézt skutečnou (využitelnou) implementaci této knihovny.

Ladící nástroje

Sebelepší vývojář na libovolné platformě je při své práci nutně autorem logických, syntaktických a mnoha dalších druhů chyb. Přitom ladění klasických GPGPU aplikací bylo a dosud je poměrně problematické. Při programování CPU aplikací jsme zvyklí používat breakpointy, krokování, sledování obsahu proměnných a další funkce ladících nástrojů, to vše s minimálním zásahem do samotného kódu aplikace. Smysluplným požadavkem tedy je existence takového nástroje pro ladění GPU aplikací. Ten by měl podporovat obě hlavní grafické API, specifická HW rozšíření jednotlivých výrobců a hlavně by měl laděný kód spouštět přímo na HW, nikoliv v nějakém emulovaném softwarovém prostředí. Ačkoliv se to možná na první pohled nezdá, některé z uvedených požadavků jsou v případě GPU poměrně problematické. Každému z existujících ladících nástrojů tak typicky chybí nějaká důležitá vlastnost (většinou spíše celá řada vlastností).

Nejpoužívanějším způsobem ladění na CPU v jednoduchých situacích je použití funkce jako je `printf` pro zobrazení obsahu určité proměnné, hodnot v poli atd. Podobně při programování GPU nám často stačí nechat si zobrazit grafická data (například obsah dynamicky se měnící textury v určitém kroku) a taková informace už postačuje k odstranění mnoha druhů chyb. Jednoduché nástroje jako je *The Image Debugger* umožňují zautomatizovat takovýto způsob získávání informací, neposkytují však žádné zvláštní možnosti pro ladění v rámci grafického API nebo ladění shaderů. Program *gDEBugger* firmy Graphic Remedy je velmi propracovaný nástroj pro ladění OpenGL programů, umožňuje v rámci OpenGL nastavovat breakpointy a krokovat, sledovat obsah stavových proměnných, sledovat obsahy textur, poskytuje mnoho funkcí pro měření výkonu a umožňuje za běhu upravovat a znovu kompilovat kód shaderů. gDEBugger je však omezen pouze na OpenGL a co je horší, nemá implementovánu podporu pro ladění a krokování samotných shaderů (můžeme tedy sledovat vstupy a výstupy vertex a fragment programů, ne však jejich průběh). Posledním přímo zmíněným nástrojem bude *Miscrosoft Shader Debugger*, ten je implementován v masově používaném Visual Studio IDE a obsahuje podporu pro přímé ladění shaderů (breakpointy, sledování proměnných, atp.). V ladícím režimu však vyžaduje, aby vlastní kód shaderů byl vykonáván v softwarovém emulovaném prostředí místo v samotném HW, což znevěrohodňuje mnohé výsledky. Podobným spůzobem bychom mohli pokračovat a zmínit další ladící nástroje a aplikace, těžko bychom však hledali takový, který by splňoval všechny naše požadavky. Více informací o zmíněných programech lze nalézt ve zdrojích v příloze A, několik dalších ladících nástrojů je uvedeno v [3].

Kapitola 2

GPGPU aplikace

Dosud jsme se poměrně podrobně zabývali popisem architektury GPU a různými technikami, které nám tato architektura dovoluje a které jsou využitelné pro GPGPU. Nyní přišel čas pokusit se spojit uvedené myšlenky dohromady a říci si něco o skutečných GPGPU aplikacích. V této kapitole zavedeme pro snadnější orientaci čtenáře číslování většiny výpisů kódu a pseudo-kódu, což v kapitole minulé nebylo třeba.

Není možné přesně vymezit oblasti problémů, mapovatelných na GPU. Jakákoliv úloha, jejíž řešení dokážeme navrhnout v rámci popsaného programovacího modelu, za splnění popsaných omezení, bude částečně nebo zcela řešitelná na GPU. Důležité je potom studium efektivity takového řešení. Přestože první pokusy o využití grafického HW (v tomto případě se samozřejmě nejednalo o grafické karty osobních počítačů) lze pozorovat zhruba v osmdesátých letech dvacátého století, skutečný nástup GPGPU do povědomí širší akademické veřejnosti je starý pouze několik málo let. Za tu dobu se objevily stovky různých aplikací a jejich počet stále prudce stoupá spolu s novými funkcemi a možnostmi vyvíjeného HW. Na tuto skutečnost už reagují i výrobci, pro něž se otevřel velmi zajímavý a zcela nový trh[1]. Numerická lineární algebra, řešení diferenciálních rovnic (ODE i PDE), fyzikální simulace, zpracování obrazu (např. segmentace), analýza signálů (existují velmi efektivní implementace rychlé Fourierovy transformace na GPU), osvětlovací techniky 3D scény (sledování paprsků, radiosita) - to jsou jen hlavní oblasti problémů efektivně řešených na GPU. Není možné ani žádoucí aby se podrobnější přehled vyskytoval v této práci, čtenáře proto odkazujeme na vyčerpávající přehled [3].

2.1 Rovnice vedení tepla

2.1.1 Stručný popis a návrh algoritmu

Uvažujme nejjednodušší parabolickou parciální diferenciální rovnici, totiž známou (Laplaceovu) *rovnici vedení tepla*

$$\frac{\partial u}{\partial t} = k\Delta u \qquad \text{na} \qquad \Omega \times (0,T) \tag{2.1}$$

kde $\Omega \subset \mathbb{R}^2$, $\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$ (Δ je Laplaceův operátor). Pro jednoduchost dále uvažujme *Dirichletovu okrajovou podmínku*

$$u|_{\partial\Omega} = \gamma,\ \gamma : \partial\Omega \longrightarrow \mathbb{R} \tag{2.2}$$

a počáteční podmínku

$$u(\cdot, 0) = u_0 \tag{2.3}$$

Zvolíme obdélníkovou oblast $\Omega \equiv (0, L_1) \times (0, L_2)$. Popsaná zjednodušení nejsou v našem případě na škodu, protože řešení tohoto problému analytickými i numerickými metodami je dobře známo.

[1] Určité předpovědi a pohled do budoucna si ponecháme na závěr této práce.

Naší snahou je implementovat popsaný problém jako GPGPU aplikaci a popsat těžkosti spojené s takovou implementací. Kvůli výpočetní náročnosti (která je hlavní motivací pro paralelizaci a použití GPGPU) i kvůli případné budoucí volbě složitějšího eliptického diferenciálního operátoru zvolíme pro řešení této úlohy metodu přímek. Provedeme tedy dvoufázovou diskretizaci nejdříve v prostoru, poté v čase. Pro prostorovou diskretizaci použijeme metodu konečných diferencí, pro diskretizaci časovou potom Runge-Kuttovu metodu 4. řádu. Na zvolenou oblast tedy položíme obdélníkovou síť zavedením prostorových kroků $h_1 = \frac{L_1}{N_1}$, $h_2 = \frac{L_2}{N_2}$, potom

$$\begin{aligned} \omega_h &= \{(ih_1, ih_2) \mid i = 1 \dots N_1 - 1, j = 1 \dots N_2 - 1\} \\ \overline{\omega}_h &= \{(ih_1, ih_2) \mid i = 0 \dots N_1, j = 0 \dots N_2\} \end{aligned}$$

Pro libovolnou funkci $u : \Omega \longrightarrow \mathbb{R}$ definujeme její projekci na síť $\overline{\omega}_h$ vztahem $u^h_{ij} = u(ih_1, jh_2)$ (síťovou funkci budeme značit symbolem h v horním indexu). V případě rovnice vedení tepla nás zajímají pouze náhrady druhých nesmíšených derivací, ty zavedeme vztahem

$$u_{\overline{x}_1 x_1, ij} = \frac{u_{i+1,j} - 2u_{ij} + u_{i-1,j}}{h_1} \ , \ u_{\overline{x}_2 x_2, ij} = \frac{u_{i,j+1} - 2u_{ij} + u_{i,j-1}}{h_2} \qquad (\text{na } \omega_h)$$

Součet uvedených dvou vztahů potom představuje diferenční náhradu Laplaceova operátoru Δ_h. Diskretizací v prostoru tedy dostaneme semidiskrétní schéma úlohy ve tvaru

$$\frac{\mathrm{d}\, u^h_{ij}}{\mathrm{d}\, t} = \Delta_h u^h_{ij} \ , \tag{2.4}$$

řešitelné standardními Runge-Kuttovými metodami (viz [20]). Pro náš případ použijeme Mersenovu variantu této metody (4. řádu), která v každém kroku provádí lokální odhad chyby a umožňuje díky jednokrokovému charakteru Runge-Kuttových metod automaticky upravovat integrační krok. Popíšeme tuto metodu pro obecnou soustavu n obyčejných diferenciálních rovnic tvaru

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}) \ , \tag{2.5}$$

které zřejmě vyhovuje i vztah 2.4[2]. V každém časovém kroku (budeme jej značit τ) je třeba napočítat koeficienty (ve skutečnosti se pro každý koeficient jedná o pole délky n)

$$\begin{aligned} \boldsymbol{K}_1 &= \tau \boldsymbol{f}(t, \boldsymbol{x}) \\ \boldsymbol{K}_2 &= \tau \boldsymbol{f}(t + \frac{\tau}{3}, \boldsymbol{x} + \frac{1}{3}\boldsymbol{K}_1) \\ \boldsymbol{K}_3 &= \tau \boldsymbol{f}(t + \frac{\tau}{3}, \boldsymbol{x} + \frac{1}{6}\boldsymbol{K}_1 + \frac{1}{6}\boldsymbol{K}_2) \\ \boldsymbol{K}_4 &= \tau \boldsymbol{f}(t + \frac{\tau}{2}, \boldsymbol{x} + \frac{1}{8}\boldsymbol{K}_1 + \frac{3}{8}\boldsymbol{K}_3) \\ \boldsymbol{K}_5 &= \tau \boldsymbol{f}(t + \tau, \boldsymbol{x} + \frac{1}{2}\boldsymbol{K}_1 - \frac{3}{2}\boldsymbol{K}_3 + 2\boldsymbol{K}_4) \ , \end{aligned} \tag{2.6}$$

lokální odhad chyby

$$\boldsymbol{E} = \frac{2\boldsymbol{K}_1 - 9\boldsymbol{K}_3 + 8\boldsymbol{K}_4 - \boldsymbol{K}_5}{30} \ , \tag{2.7}$$

a v případě pozitivního testu na maximální hodnotu chyby (bude vysvětleno dále) i nové hodnoty neznámé funkce

$$\boldsymbol{x}(t + \tau) = \boldsymbol{x}(t) + \frac{1}{6}\boldsymbol{K}_1 + \frac{2}{3}\boldsymbol{K}_3 + \frac{1}{6}\boldsymbol{K}_5 \tag{2.8}$$

[2]Jiný eliptický diferenciální operátor by po diskretizaci dal jinou pravou stranu ve vztazích 2.4 resp. 2.5, podstata Runge-Kuttovy-Mersenovy metody by však zůstala zachována.

Jak již bylo naznačeno, odhad chyby se používá pro stanovení nové hodnoty časového kroku pro další výpočty a v tomto smyslu je celý algoritmus adaptivní. Nová hodnota časového kroku se počítá ze vztahu

$$\tau = \omega\tau \left(\frac{\varepsilon}{\vartheta}\right)^{\frac{1}{5}} ,$$

kde ε je maximální dovolená hodnota chyby (konstantní parametr), ω se obvykle volí jako konstantní parametr z intervalu $\langle 0.8, 0.9\rangle$ a ϑ představuje maximální vypočítanou hodnotu chyby, danou vztahem $\vartheta = \max|\boldsymbol{E}|$. Nyní máme připraven nutný minimální matematický aparát a celý algoritmus můžeme popsat následujícím pseudokódem, přičemž volané funkce přesně odpovídají uvedeným vzorcům:

```
while(t_aktual < t_konec) {
    //Výpočet Runge–Kuttových koeficientů
    K1 = vypocti_K1(u);
    K2 = vypocti_K2(u, K1);
    K3 = vypocti_K3(u, K1, K2);
    K4 = vypocti_K4(u, K1, K3);
    K5 = vypocti_K5(u, K1, K3, K4);
    //Odhad chyby
    E = vypocet_chyby(K1, K3, K4, K5);
    Emax = max(abs(E));
    if (Emax < epsilon) {
        //Nové hodnoty nezmáné funkce v čase t_aktual + tau
        u = aktualizuj_reseni(u, K1, K3, K5);
        t_aktual += tau;
    }
    tau = aktualizace_cas_kroku(tau);
}
```

Výpis 2.1: Algoritmus Runge-Kuttovy-Mersenovy metody

Nyní si ještě o něco málo zjednodušíme situaci a budeme předpokládat $L_1 = L_2 = 1$, $N_1 = N_2 = N = 2^k$ pro nějaké $k \in \mathbb{N}$, $\gamma = 0$ (nulová okrajová podmínka), pak tedy Ω bude představovat jednotkový čtverec pokrytý pravidelnou čtvercovou (regulární) sítí, dále uvidíme jak nám toto zjednodušení pomůže.

V dalších úvahách se opět věnujme popsanému algoritmu v rámci vztahu 2.4. Všímavý čtenář si jistě uvědomí, že tento algoritmus je velmi dobře paralelizovatelný (dokonce několika způsoby). Jestliže máme k dispozici dostatek procesorů, mohou tyto nezávisle paralelně počítat jednotlivé elementy pole `K1`, budou-li mít přístup do sdílené paměti se vstupním polem `u`. Podobně lze počítat ostatní Runge-Kuttovy koeficienty, pole chyb `E`, i průběžně aktualizovat řešení. A co je ještě důležitější, uvedený algoritmus lze díky popsané nezávisloti snadno „nasadit“ na proudový programovací model, popsaný v části 1.2.3. Aby toto bylo úplně zřejmé, prohlédněme si následující výpis:

```
while(t_aktual < t_konec) {
    //Výpočet Runge–Kuttových koeficientů
    K1 << u;
    K2 << (u, K1);
    K3 << (u, K1, K2);
    K4 << (u, K1, K3);
    K5 << (u, K1, K3, K4);
    //Odhad chyby
    E << (K1, K3, K4, K5);
    Emax <<red E;
```

```
    if (Emax < epsilon) {
        //Nové hodnoty nezmáné funkce v čase t_aktual + tau
        u << (u, K1, K3, K5);
        t_aktual += tau;
    }
    tau = aktualizace_cas_kroku(tau);
}
```

Výpis 2.2: Algoritmus Runge-Kuttovy-Mersenovy metody s naznačením proudového zpracování

V něm jsme použili pseudo-operátor `<<`, který z jednoho nebo více proudů na pravé straně vygeneruje jeden nebo více proudů na straně levé[3]. Podobně pseudo-operátor `<<red` provádí na vstupním proudu operaci *redukce*, výstupním proudem je potom jediné číslo (konkrétně hledané maximum absolutních hodnot). Uvedený zápis je značně abstraktní, použité pseudo-operátory nemají nic společného s operátory jazyka C++. Autor se tímto zápisem pouze snaží ulehčit čtenáři myšlenkový skok k popisovanému programovacímu modelu. Nyní konečně můžeme říci, že každý z řádků obsahující nějaký proudový operátor bude moci být implementován na GPU a výpočet proběhne jedním nebo více průchody grafickou pipeline. Celý problém tak půjde implementovat jako (mnohaprůchodová) GPGPU aplikace. Na důležité detaily takové implementace se podíváme v další části.

2.1.2 GPGPU implementace

Jak bylo popsáno v části 1.3.3, pochopení jednoduché GPGPU aplikace otevírá prakticky dokořán dveře k tvorbě aplikací složitějších. Konfigurace grafické pipeline i komunikace s runtime knihovnou jazyka Cg bude probíhat přesně stejným způsobem. Jen budeme v tomto případě potřebovat více textur pro uložení mezivýsledků, více FBO pro renderování do těchto textur a více fragment programů (shaderů), které budou vystupovat v roli jader na proudech a provádět na nich požadované aritmetické operace.

Z uvedených algoritmů je vidět, že nejčastěji prováděnou operací na proudu dat zřejmě bude výpočet pravé strany ze vztahu 2.4, totiž výpočet diskrétního Laplaceova operátoru. Implementujeme jej následujícím fragment programem, který na vstupní textuře (resp. vstupním proudu resp. vstupním poli) `tex` provede právě požadovanou operaci.

```
struct SInput {
    uniform float tau;
    uniform float2 dxdy2;
    uniform samplerRECT tex;
    float2 crd : TEXCOORD0;
};

//Funkce s návratovým typem float, realizující vlastní výpočet
float dLaplace(samplerRECT data, float2 crd, float2 dxdy2) {
    float u_i_j = texRECT(data, crd).r;
    float u_im_j = texRECT(data, crd + float2(-1,0)).r;
    float u_ip_j = texRECT(data, crd + float2(1,0)).r;
    float u_i_jm = texRECT(data, crd + float2(0,-1)).r;
    float u_i_jp = texRECT(data, crd + float2(0,1)).r;
    float der2x = (u_ip_j - 2*u_i_j + u_im_j)/(dxdy2.x);
    float der2y = (u_i_jp - 2*u_i_j + u_i_jm)/(dxdy2.y);
```

[3]V tomto místě zanedbáváme způsob, jakým toto zpracování proudů probíhá. Zpracování proudů (jádro), které skrývá operátor `<<`, se samozřejmě řádek od řádku liší, základní princip vyhovující proudovému programovacímu modelu však zůstává stále stejný.

```
    return der2x + der2y;
}

float main(SInput inData) : COLOR {
    float outData = inData.tau * dLaplace(inData.tex, inData.crd, inData.dxdy2);
    return outData;
}
```

Výpis 2.3: Fragment program implementující diskrétní Laplaceův operátor

Tento fragment program bude v daném časovém kroku volán opakovaně a v pěti průchodech postupně získáme Runge-Kuttovy koeficienty `K1`,...,`K5`. Z algoritmu 2.1 a předchozích vzorců je dále vidět, že velmi často používanou operací na proudech dat bude jistá zobecněná forma operace $SAXPY$[4]. Konkrétně potřebujeme určitý počet vektorů (polí) vynásobit konstantou a poté sečíst, jak je vidět například ze vztahu 2.7. Takovou operaci implementuje triviální fragment program z následujícího výpisu:

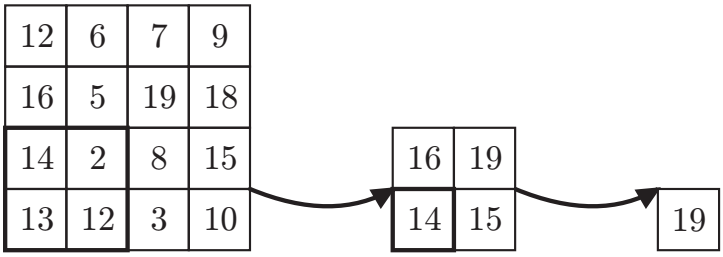
```
struct SInput {
    uniform samplerRECT texK1;
    uniform samplerRECT texK3;
    uniform samplerRECT texK4;
    uniform samplerRECT texK5;
    float2 crd : TEXCOORD0;
};

float main(SInput in) : COLOR {
    float outData = (2.0/30.0)*texRECT(inData.texK1, inData.crd)
        - (9.0/30.0)*texRECT(inData.texK3, inData.crd)
        + (8.0/30.0)*texRECT(inData.texK4, inData.crd)
        - (1.0/30.0)*texRECT(inData.texK5, inData.crd);
    return out;
}
```

Výpis 2.4: Fragment program pro výpočet lokálního odhadu chyby Runge-Kuttovy metody

Pro shadery uvedeného typu je třeba mít na paměti jediné omezení, související s počtem texturovacích jednotek daného HW. Dnešní GPU umožňují na jeden polygon mapovat určité množství textur (dáno např. konstantou `MAX_TEXTURE_IMAGE_UNITS_ARB`), a určité množství souřadnic textur (dáno např. konstantou `GL_MAX_TEXTURE_COORDS_ARB`). Potřebujeme-li sečíst více vektorů, než umožňuje konstanta `MAX_TEXTURE_IMAGE_UNITS_ARB`, musíme tak učinit víceprůchodovým algoritmem.

Netriviální přístup však musíme použít pro implementaci operace redukce, potřebujeme totiž maximum absolutních hodnot z napočítaných chyb (viz výpis 2.1). Naivní přístup by mohl znít renderovat čtverec o rozměru 1×1 (vzniknul by tak jediný fragment) a spustit fragment program, který projde všechny texely vstupní textury a na nich provede redukci, podobně jako bychom ji implementovali na CPU. V takovém případě by však veškerou práci dělal jediný FP a nedošlo by tak k žádné paralelizaci. Jak tedy provést takovou operaci paralelně v rámci proudového programovacího modelu? Řešení naštěstí vůbec není složité, spočívá v „pyramidální“ redukci problému. Mějme vstupní pole dat rozměru $N \times N$. Jednotlivé elementy rozdělme do symetrických bloků rozměru 2×2. Takové bloky potom můžeme nezávisle paralelně zpracovávat, na každém z nich provede procesor redukci a do výstupního pole uloží jediný element. Výstupem bude pole dat o poloviční rozměrech a dále se celý

[4]Zkratka $SAXPY$ pochází z angl. Scalar Alpha X Plus Y, je často pužívána v počítačové literatuře. Jedná se o realizaci základní vektorové operace lineární algebry, totiž operace $\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$. Tato operace je jako jedna z elementárních implementována např. ve známé knihovně BLAS.



Obrázek 2.1: Operace redukce na GPU

proces opakuje. K celkové redukci vstupního pole potřebujeme $\log N$ takových kroků. Je vidět, že takový přístup velmi dobře odpovídá proudovému programovacímu modelu a půjde tedy implementovat na GPU. Celý postup je znázorněn na obrázku 2.1.

Pro celkovou redukci je třeba $\log N$ renderovacích kroků, v každém kroku renderujeme obdélník o poloviční rozměrech, než má vstupní textura (vstupní pole dat), renderovaný obdélník je na obrázku 2.1 vyznačen tučně. V tomto bodě je vidět smysl zavedení dodatečné podmínky $N_1 = N_2 = N = 2^k$ pro nějaké $k \in \mathbb{N}$, díky ní je totiž $\log N \in \mathbb{N}$ a uvedená redukce tak proběhne přirozeným způsobem, aniž bychom se museli starat o nějaké doplňování vstupního pole na správný rozměr nebo hledání alternativního řešení. Jak již bylo řečeno, v každém kroku redukce renderujeme geometrii o poloviční rozměrech. Přitom existují dva způsoby, jak vrcholům přiřadit správné souřadnice textury:

1. Přiřadit tyto souřadnice vrcholům při vlastní definici geometrie na úrovni OpenGL API, přičemž o jejich spočítání na úrovni jednotlivých fragmentů se postará interpolátor
2. Spočítat souřadnice za běhu v samotném fragment programu

Tato práce se opírá o první uvedený způsob, jeho realizace je uvedena na následujícím výpisu:

```
glBegin(GL_QUADS);
glMultiTexCoord2f(GL_TEXTURE0, -0.5, -0.5);
glMultiTexCoord2f(GL_TEXTURE1, 0.5, -0.5);
glMultiTexCoord2f(GL_TEXTURE2, 0.5, 0.5);
glMultiTexCoord2f(GL_TEXTURE3, -0.5, 0.5);
glVertex2i(0,0);

glMultiTexCoord2f(GL_TEXTURE0, (2*width)-0.5, -0.5);
glMultiTexCoord2f(GL_TEXTURE1, (2*width)+0.5, -0.5);
glMultiTexCoord2f(GL_TEXTURE2, (2*width)+0.5, 0.5);
glMultiTexCoord2f(GL_TEXTURE3, (2*width)-0.5, 0.5);
glVertex2i(width,0);

glMultiTexCoord2f(GL_TEXTURE0, (2*width)-0.5, (2*height)-0.5);
glMultiTexCoord2f(GL_TEXTURE1, (2*width)+0.5, (2*height)-0.5);
glMultiTexCoord2f(GL_TEXTURE2, (2*width)+0.5, (2*height)+0.5);
glMultiTexCoord2f(GL_TEXTURE3, (2*width)-0.5, (2*height)+0.5);
glVertex2i(width,height);

glMultiTexCoord2f(GL_TEXTURE0, -0.5, (2*height)-0.5);
glMultiTexCoord2f(GL_TEXTURE1, 0.5, (2*height)-0.5);
glMultiTexCoord2f(GL_TEXTURE2, 0.5, (2*height)+0.5);
```

```
glMultiTexCoord2f(GL_TEXTURE3, -0.5, (2*height)+0.5);
glVertex2i(0,height);
```

Výpis 2.5: Renderování geometrie pro operaci redukce

Každému vrcholu vstupní geometrie jsme tak přiřadili čtveřici souřadnic textury (a tím i 4 různé texely), v souladu s popsaným postupem. O vlastní výpočet maxima z absolutních hodnot už se potom snadno postará následující fragment program:

```
struct SInput {
    //Vstupní informací jsou pro každý fragment 4 odlišné souřadnice textury
    float2 ll: TEXCOORD0;
    float2 lr: TEXCOORD1;
    float2 ur: TEXCOORD2;
    float2 ul: TEXCOORD3;
    uniform samplerRECT tex;
};

float main(SInput inData) : COLOR {
    //Načtení dat z textur a uložení absolutních hodnot
    float data1 = abs((float)texRECT(inData.tex, inData.ll));
    float data2 = abs((float)texRECT(inData.tex, inData.lr));
    float data3 = abs((float)texRECT(inData.tex, inData.ur));
    float data4 = abs((float)texRECT(inData.tex, inData.ul));
    //Vlastní výpočet maxima
    float outData = max(data4,max(data3,max(data2,data1)));
    return outData;
}
```

Výpis 2.6: Fragment program realizující operaci redukce - výpočet maxima z absolutních hodnot

Ping pong technika

V části 1.3.1 jsme uvedli, že v principu nelze mít jednu texturu připojenu pro čtení a jako cíl renderovací operace zároveň. V naprosté většině GPGPU algoritmů se tak využívá tzv. *ping pong technika*. Jedná se o populární označení velmi jednoduchého principu. V poslední fázi daného časového kroku potřebujeme aktualizovat funkční hodnoty z času t na funkční hodnoty, odpovídající času $t + \tau$ (viz. vztah 2.8). Pro řešení takového vztahu však potřebujeme dvě textury (při nevektorizovaném výpočtu na CPU by nám stačilo jediné pole s daty), jedna slouží jako zdroj dat, druhá jako cíl renderovací operace, v následujícím časovém kroku si pak obě textury „prohodí úlohy“. Celý princip (pro jakýkoliv obecný výpočet) je popsán v následujícím výpisu, který obsahuje kombinaci kódu a pseudo-kódu (pro připomenutí volaných funkcí odkazujeme čtenáře zpět na 1.3.3):

```
//Tyto proměnné nám budou sloužit jako ukazatele na textury
int readTex = 0;
int writeTex = 1;
//Použijeme dvě textury a dva přípojné body
GLuint tex[2];
GLenum attachpoints = {GL_COLOR_ATTACHMENT0_EXT, GL_COLOR_ATTACHMENT1_EXT};
...
//Zde proběhne inicializace textur tex[0] a tex[1] na straně GPU,
//dále inicializace FBO, předpokládejme že textury jsou připojeny k FBO způsobem:
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
```

```
        GL_COLOR_ATTACHMENT0_EXT, texTarget, tex[0], 0);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
        GL_COLOR_ATTACHMENT1_EXT, texTarget, tex[1], 0);
//Zde proběhnou další inicializace a nastavení před vlastním renderováním
...
while(dokud_nemame_vysledek) {
    //Určíme texturu tex[writeTex] jako cíl renderovací operace
    glDrawBuffer(attachpoints[writeTex]);
    //Následuje vlastní výpočet na GPU
    pripoj_texturu_pro_cteni(tex[readTex]);
    proved_vypocet()
    //Nakonec si obě textury prohodí role
    swap(readTex, writeTex);
}
```

Výpis 2.7: Pseudokód popisující ping pong techniku

Uvedený výpis jistě nepředstavuje jedinou možnou implementaci, princip techniky by však měl být v tuto chvíli zřejmý. V popisované aplikaci nám použití dvou různých textur pro vstup resp. výstup nevadilo a kromě potřeby uvažovat dvojnásobné množství paměti nepřinášelo další komplikace[5]. V následující aplikaci však uvidíme, že proudový model a nutnost použití dvou různých textur nám někdy zasáhnou do samotné logiky algoritmu.

2.1.3 Shrnutí, hodnocení a závěr

Ostatní části popisované aplikace jsou pouze rozšířením toho, co bylo uvedeno v demonstračním příkladu v části 1.3.3. Na obrázku 2.2 je uveden časový vývoj počáteční podmínky u_0 na oblasti $\Omega \equiv (0,1)^2$ pokryté sítí 512×512, uvažujeme Dirichletovu okrajovou podmínku $u|_{\partial\Omega} = 0$ a počáteční podmínku (kružnice o poloměru $\frac{1}{4}$, posunutá do středu uvažované oblasti):

$$u_0 = \begin{cases} 1 & \text{pro } (x-\frac{1}{2})^2 + (y-\frac{1}{2})^2 \leq (\frac{1}{4})^2 \\ 0 & \text{pro } (x-\frac{1}{2})^2 + (y-\frac{1}{2})^2 > (\frac{1}{4})^2 \end{cases}$$

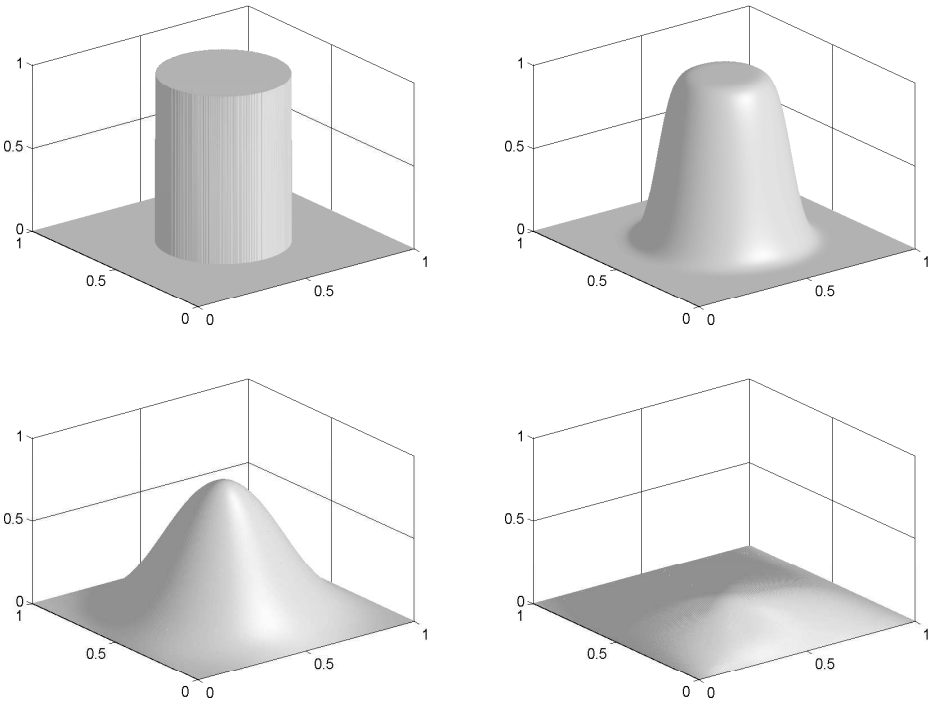
Výpočet byl prováděn do konečného času $T = 0.1$. Uvedené obrázky samozřejmě nejsou překvapivé, řešení rovnice vedení tepla bylo již nesčetněkrát zpracováno, co však můžem říci přímo o GPGPU implementaci? V této práci jsme se soustředili na vlastní studium možností implementace a její realizaci. Z popsaných důvodů se autor prozatím nepokoušel o relevantní měření a srovnávání výkonu, přestože předběžné výsledky naznačují, že GPU implementace by mohla oproti ekvivalentnímu CPU řešení vykazovat několikanásobné zrychlení. Naše implementace zahrnuje celkem 8 fragment programů:

- 1 na výpočet Laplaceova operátoru, nazvaný `laplace`
- 4 pro mezivýpočty před aplikací pravé strany (viz. vztahy 2.6), nazvané `pre_k2`, `pre_k3`, `pre_k4` a `pre_k5`
- 1 pro výpočet lokálních odhadů chyby (viz. vztah 2.7), nazvaný `error`
- 1 pro implementaci operace redukce (maximum z absolutních hodnot), nazvaný `max_abs`
- 1 pro aktualizaci funkčních hodnot (viz. vztah 2.8), nazvaný `update`

Všechny fragment programy sekvenčně zpracovávají hodnoty síťových funkcí a vykazují dobrou lokalitu dat ve 2D[6] (jedná se tedy o efektivní implementaci vzhledem k popsané organizaci cache paměti).

[5] Uvedená poznámka o paměti není zcela relevantní, při výpočtech na GPU musíme vždy uvažovat framebuffer + požadované množství textur s daty. V metodě renderování do textury pouze nahradíme framebuffer texturou. Z toho důvodu nelze zcela srovnávat paměťové nároky algoritmu na GPU a ekvivalentního CPU algoritmu.

[6] Každý fragment přistupuje k „vlastnímu texelu“, který mu byl přiřazen interpolací souřadnic textury, případně k texelům v nějakém malém okolí, jak je tomu např. v případě fragment programu `laplace`.



Obrázek 2.2: Rovnice vedení tepla, vývoj počáteční podmínky v časech $t = 0$, $t = 0.001$, $t = 0.01$, $t = 0.1$

Co můžeme říci o výpočetní náročnosti a aritmetické intenzitě? Fragment programy `pre_k2`, `pre_k3`, `pre_k4`, `pre_k5`, `error` a `update` jsou všechny stejného typu a obsahují maximálně $(tex - 1) + 1$ (kde tex je počet instrukcí čtení z textury) aritmetických intstrukcí ($tex - 1$ instrukcí MAD plus jedno případné násobení). To odpovídá aritmetické intenzitě nejvýše rovné dvěma. Program `max_abs` je tvořen $tex - 1$ instrukcemi pro výpočet maxima. Největší počet aritmetických výpočtů obsahuje program `laplace` (přesný počet instrukcí závisí na úrovni optimalizace ze strany překladače a některých detailních vlastnostech GPU, nebudeme jej zde proto uvádět), ačkoliv i v jeho případě nepřekračuje aritmetická intenzita o mnoho číslo 2. Celá aplikace tak bude omezena spíše přenosovou rychlostí paměti než maximálním výpočetním výkonem GPU (což však vzhledem k faktům popsaným v části 1.4.1 nesnižuje „vhodnost“ implementace tohoto problému na GPU).

Uvedené fragment programy jsou implementovány skalárně, vzhledem k uložení dat v jednosložkové textuře typu `LUMINANCE` jsme prozatím nenalezli způsob, jak efektivně využít instrukční paralelismus a popsaný vektorový charakter FP. Většina klasických článků o GPGPU doporučuje vektorizaci a využití čtyřsložkových textur typu RGBA, bylo by tedy jistě zajímavé pokusit se o vektorizaci tohoto problému [7]. Popišme si nyní ještě jednu velmi přirozenou aplikaci GPGPU, tentokrát půjde o úlohu z lineární algebry.

2.2 LU rozklad husté matice

Jak jsme se již zmínili, GPGPU se velice hodí k urychlováním některých operací lineární algebry. Vždyť už tak jednoduchá operace, jako je SAXPY může být díky nezávislosti a šířce paměťové sběrnice

[7] Na druhou stranu nově nastupující GPU s podporou DX10 nají v HW implementovány *skalární* programovatelné procesory, což činí vhodnost vektorizace více než diskutabilní. Více informací nalezne čtenář v poslední kapitole této práce.

efektivně paralelně počítána na GPU (viz. úvahy v části 1.5.1). Tomu, aby GPU mohl být využit jako koprocesor pro akcelerování takových operací, však brání omezená přenosová rychlost mezi CPU $\longleftrightarrow$ GPU. V této části tak zaměříme svoji pozornost na jiný (o něco málo složitější) problém z lineární algebry, totiž hledání LU rozkladu husté matice. Víme, že složitost algoritmů řešících tento problém je $O(n^3)$, takže výpočetní náročnost bude pro matice dostatečných rozměrů amortizovat latence spojené s přenosem dat z hostitelské paměti na GPU. Nejdříve krátce připomeneme známou formální stránku našeho problému.

Pro $\boldsymbol{A} \in \mathbb{R}^{n,n}$, $n \in \mathbb{N}$, hledáme rozklad

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}, \tag{2.9}$$

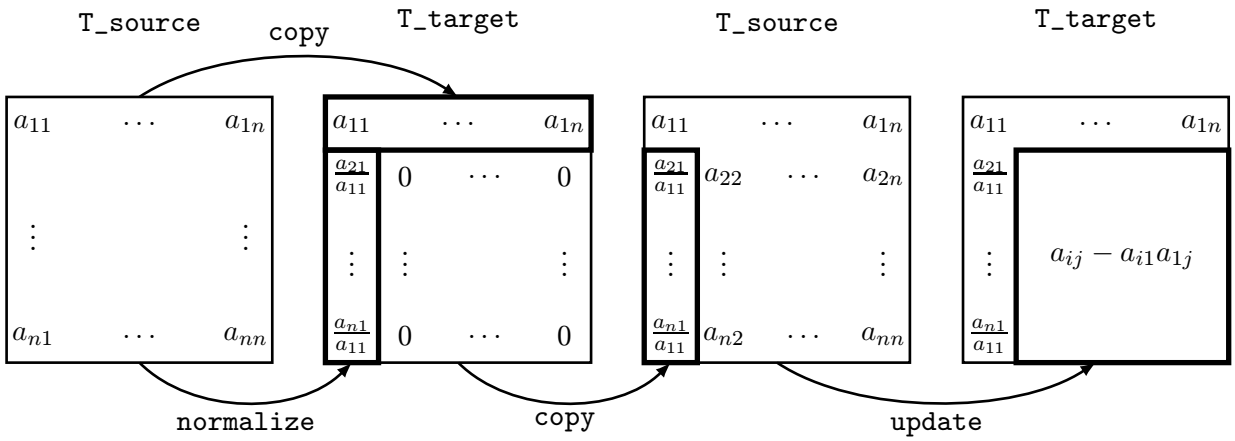
kde $\boldsymbol{L}$ je dolní trojúhelníková matice s jedničkami na diagonále a $\boldsymbol{U}$ je horní trojúhelníková matice. Takový rozklad lze potom jednoduše použít k řešení soustavy lineárních algebraických rovnic, ve skutečnosti se jedná o přístup ekvivalentní (ve známém smyslu) Gaussově eliminaci. Nutnou a postačující podmínkou existence a jednoznačnosti rozkladu je silná regularita matice $\boldsymbol{A}$ (tzn. $\forall k \in \{1, \ldots, n\}(\det \boldsymbol{A}(1{:}k, 1{:}k) \neq 0)$), důkaz tohoto tvrzení a další informace je možné nalézt například v [21]. Také víme, že každou regulární matici lze vhodnou permutací řádků a sloupců přeuspořádat na matici silně regulární. Problémem je samozřejmě hledání takových permutací. Protože v obecném případě nemáme zaručenu existenci rozkladu ani jeho stabilitu, používá se ve skutečných aplikacích částečný nebo úplný *pivoting*. My se však zaměříme na nejjednodušší verzi algoritmu bez pivotingu, složitější verze je možné nalézt v [19]. Uveďme nyní klasickou podobu algoritmu:

```
for k = 1,...,n-1 {
    for i = k+1,...,n {
        a(i,k) = a(i,k) / a(k,k);
    }
    for i = k+1,...,n {
        for j = k+1,...,n {
            a(i,j) = a(i,j) - a(i,k)*a(k,j);
        }
    }
}
```

Výpis 2.8: Tzv. submatická metoda(k-i-j metoda) na LU rozklad matice

CPU implementace takového algoritmu je velice přímočará, veškeré operace probíhají v jednom dvourozměrném poli, v něm také nakonec nalezneme požadovaný rozklad. Pro GPGPU implementaci však musíme použít popsanou ping-pong techniku a dvě textury (připojené ke dvěma různým povrchům FBO jakožto offscreen bufferu). Snažit se popsat hledaný algoritmus způsobem podobným výpisu 2.2 by čtenáře spíše zmátlo, proto si v tomto případě pomůžeme obrázkem a slovním popisem. V každém z $n-1$ kroků vnější smyčky potřebujeme provést dvě základní operace - normalizaci k-tého sloupce matice $\boldsymbol{A}$, poté úpravu submatice $\boldsymbol{A}(k+1{:}n, k+1{:}n)$. Obě operace jsou zřejmě nezávisle paralelizovatelné a půjdou snadno implementovat na GPU, o normalizaci se postará fragment program `normalize`, aplikovaný na fragmenty odpovídající příslušnému sloupci matice, úpravu submatice $\boldsymbol{A}(k+1{:}n, k+1{:}n)$ zajistí fragment program `update`. Ve skutečnosti budeme muset použít ještě jeden fragment program nazvaný `copy`, který v souladu se svým názvem pouze kopíruje vstupní hodnotu na výstup[8]. Toto kopírování nijak nesouvisí s původním algoritmem, jedná se o určité „nutné zlo“ související s použitím dvou různých datových polí (resp. dvou různých textur, resp. dvou různých renderovacích povrchů). Provedení prvního kroku (pro $k = 1$) LU rozkladu na GPU je naznačeno na obrázku 2.3, algoritmus celé implementace je uveden ve výpisu 2.9. Pro zjednodušení označujeme zdrojovou resp. cílovou texturu i k ní připojený renderovací povrch jako `T_source` resp. `T_target`, v

[8] jelikož používáme metoru RTT, můžeme za vstup i výstup považovat přímo texely textury.



Obrázek 2.3: První krok LU rozkladu husté matice na GPU

tuto chvíli nám taková nejednoznačnost nebude na obtíž. Funkce `swap(T_source,T_target)` z výpisu tak vyjadřuje algoritmický požadavek na prohození, její skutečná implementace je řešena podobně, jak bylo uvedeno v předcházející části při popisu ping pong techniky.

```
for k = 1,...,n-1 {
    //Kopírování k−tého řádku do cílové textury
    glDrawBuffer(T_target);
    Připoj texturu T_source pro čtení;
    load_fragment_program(copy);
    Renderuj obdélník pokrývající k-tý řádek;
    //Normalizace k−tého sloupce
    load_fragment_program(normalize);
    Renderuj obdélník pokrývající k-tý sloupec;
    //Kopie normalizovaného sloupce zpět do zdrojové textury
    glDrawBuffer(T_source);
    Připoj texturu T_target pro čtení;
    load_fragment_program(copy);
    Renderuj obdélník pokrývající k-tý sloupec;
    //Aktualizace příslušné submatice
    glDrawBuffer(T_target);
    Připoj texturu T_source pro čtení;
    load_fragment_program(update);
    Renderuj obdélník pokrývající submatici A(k+1:n,k+1:n);
    //Nakonec je třeba prohodit význam textur(renderovacích povrchů) T_source a T_target
    swap(T_source,T_target);
}
```

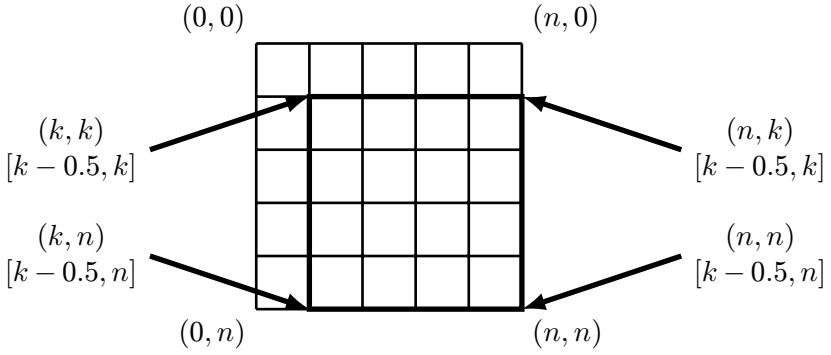
Výpis 2.9: GPGPU implementace LU rozkladu

K-tý krok uvedeného algoritmu tedy zahrnuje čtyři průchody grafickou pipeline (čtyři renderovací kroky), po nichž v `T_target` najdeme průběžný výsledek rozkladu. Nezbývá nám, než si říci něco o implementaci uvedených fragment programů a několik obecných poznámek ke GPGPU implementaci jako takové. První z těchto poznámek se týká souřadných systémů. Souřadné systémy obrazové roviny resp. textury jsou v OpenGL specifikovány takto:

$$\begin{array}{ccc} y & & t \\ \uparrow & \text{resp.} & \uparrow \\ \rightarrow\ x & & \rightarrow\ s \end{array}$$

Mějme tedy texturu, která obsahuje data matice $\boldsymbol{A}$ a je namapována na obdélník příslušné velikosti. Texel $T(i,j)$ v souřadnicích textury resp. fragment $f(i,j)$ v obrazové rovině pak odpovídá elementu a_{ji} matice $\boldsymbol{A}$, musíme tedy uvažovat „transponované souřadnice“.

V GPGPU řešení rovnice vedení tepla uvedeného v části 2.1 jsme ve většině průchodů renderovali obdélník pokrývající celou množinu vstupních dat (resp. množinu vnitřních bodů sítě). V tomto případě potřebujeme provádět výpočty na submaticích (řádek, sloupec, submatice) a renderované obdélníky tak budou tyto submatice pokrývat. Fragment program `copy` není třeba diskutovat, jeho implementace je nejjednodušší možná (snadno jej dostaneme například úpravou programu uvedeného ve výpisu 2.4). Zajímavější už je implementace fragment programů `normalize` a `update` (ačkoliv výpočetně se stále jedná o jednu nebo dvě aritmetické instrukce). Program `update` by měl řešit vztah `a(i,j) = a(i,j) - a(i,k)*a(k,j)` (viz. algoritmus 2.8). Jak získat při zpracování fragmentu souřadnice (k,i) resp. (j,k) pro načtení texelů $T(k,i)$ resp. $T(j,k)$ (odpovídá elementům a_{ik} resp. a_{kj} původní matice)? Nejjednodušší možností se jeví předat hodnotu k fragment programu jakožto konstantní parametr a v těle programu potom rekonstruovat souřadnice (k,i) resp. (j,k). Přirozenější postup pro GPU (který navíc „šetří“ instrukce programu) však velí využít interpolátor a mechanizmus mapování textur (klíčová je možnost využít multitexturing a více souřadnic textur najednou). Pro úpravu submatice $\boldsymbol{A}(k+1{:}n, k+1{:}n)$ renderujeme obdélník o souřadnicích (k,k),(k,n),(n,n),(n,k). Jestliže těmto vrcholům přiřadíme souřadnice textury $[k-0.5,k]$,$[k-0.5,n]$,$[k-0.5,n]$,$[k-0.5,k]$, budeme mít díky interpolátoru při zpracování fragmentu $f(j,i)$ k dispozici souřadnici $(k-0.5,i)$, které odpovídá hledaný texel $T(k,i)$[9]. Popsaný způsob mapování textury je pro $k=1$ naznačen na obrázku 2.4[10]. Zcela ekvivalentně potom na renderovaném obdélníku specifikujeme další souřadnice textury pro určení texelu $T(j,k)$. Jesliže porozumíme uvedené technice, pak vlastní implementace



Obrázek 2.4: Mapování textury pro fragment program `update`

fragment programu `update` je snadnou záležitostí, potřebujeme načíst tři texely textury a aplikovat dvě aritmetické operace, jak je uvedeno v následujícím výpisu:

[9] Konstanta 0.5 při deklaraci souřadnic textury není vybrána náhodně, ve spojení s rovnoběžným promítáním a správně zvolenou velikostí viewportu totiž způsobí, že texturovací jednotka bude adresovat přesně střed daného texelu. Jelikož však ve všech texturách důsledně využíváme mapování typu `GL_NEAREST`(viz. 1.3.3), fungoval by uvedený postup stejně dobře se souřadnicemi textury $[k-1,k]$,$[k-1,n]$,$[k-1,n]$,$[k-1,k]$, nebo dokonce se souřadnicemi $[k-1,k]$,$[k-1,n]$,$[k,n]$,$[k,k]$. Pro dokonalé pochopení této poznámky odkazujeme čtenáře na [5].

[10] Uvedli jsme, že osa y směřuje ve specifikaci OpenGL ve směru $\uparrow$, obrázek 2.4 bychom si tedy měli představovat zrcadlově převrácený podle osy x. Autor této práce se jej však rozhodl uvést v této podobě, ve které lépe odpovídá tomu, jak obvykle zapisujeme a „adresujeme“ matice.

```
struct SInput {
    uniform samplerRECT tex;
    //Souřadnice texelu T(j,i)
    float2 crd0 : TEXCOORD0;
    //Souřadnice texelu T(k,i)
    float2 crd1 : TEXCOORD1;
    //Souřadnice texelu T(j,k)
    float2 crd2 : TEXCOORD2;
};
float main(SInput inData) : COLOR {
    float outData = texRECT(inData.tex, inData.crd0)
        - (texRECT(inData.tex, inData.crd1)*texRECT(inData.tex, inData.crd2));
    return outData;
}
```

Výpis 2.10: Fragment program `update`

Velmi podobně lze řešit implementaci fragment programu `normalize`. Opět můžeme předat souřadnici (k, k) programu jakožto konstantní parametr a použít ji k adresování textur a získání texelu $T(k, k)$, nebo využít rasterizér a uspořádanou dvojici $(k - 0.5, k - 0.5)$ specifikovat jako souřadnici textury na všech čtyřech vrcholech renderovaného obdélníka. Rasterizér potom provede interpolaci (čtyřech stejných souřadnic $(k - 0.5, k - 0.5)$, čímž vznikne samozřejmě ta samá hodnota), výsledek použijeme k získání texelu $T(k, k)$ jakožto děliteli v operaci `a(i,k) = a(i,k) / a(k,k)`. Byla provedena měření která dokazují, že použití mapování textur je ve výsledku rychlejší, než použití konstantních parametrů (ačkoliv v případě fragment programu `normalize` je toto zrychlení zanedbatelné). Použití souřadnic textur umožňuje GPU předpovídat využití texelů a načítat je předčasně do cache paměti, což samozřejmě ovlivňuje výkon.

Hodnocení a závěr

Zkusme se podívat na počty instrukcí popsaných fragment programů, program `copy` bude jistě tvořen jedinou instrukcí pro čtení dat z textury (neobsahuje tedy žádnou aritmetickou instrukci), program `normalize` tvoří jedna aritmetická intrukce DIV[11](a dvě čtení z textury), program `update` obsahuje jednu instrukci MAD(a tři čtení z textury). Všechny programy tedy vykazují aritmetickou intenzitu nejvýše rovnou jedné. Výkon výsledné implementace tak bude ovlivněn spíše propustností paměti než maximálním výpočetním výkonem GPU. Přístup do paměti je však v každém renderovacím kroku sekvenční (vzhledem k 2D organizaci paměti) a cache-koherentní, tedy optimální pro GPU (viz. popis v části 1.4.1). Ukazuje se, že rychlost GPGPU implementace LU rozkladu je velmi dobře škálovatelná s počtem fragment procesorů a je schopna překonávat optimalizované ATLAS[12] CPU implementace. GPGPU implementace navíc efektivně využívá propustnost paměti a není závislá na velikosti cache. Tyto a další měření a závěry, které nebyly součástí této práce, je možné nalézt v [19].

[11] Překladač ve skutečnosti operaci dělení optimalizuje a rozloží do dvou aritmetických instrukcí, pro výpočet převrácené hodnoty a následné násobení touto hodnotou.

[12] z angl. Automatically Tuned Linear Algebra Software, jedná se o jednu z implementací knihovny BLAS (a některých částí knihovny LAPACK), která se automaticky přizpůsobuje cílovému HW (jedna z mnoha optimalizací umožňuje generovat kód s ohledem na velikost cache paměti).

Kapitola 3

GPGPU v moderním pojetí

GPGPU se v posledních letech dostává do stále širšího povědomí odborné veřejnosti. Zatímco ještě před pár lety se jednalo o techniku určenou úzkému okruhu „nadšenců“ a vizionářů, v dnešní době se o tomto způsobu provádění paralelních výpočtů zcela běžně dočítáme v magazínech a dalších zdrojích zaměřených na HPC[1]. Tento vývoj si uvědomili i výrobci grafických čipů a dnes už berou GPGPU komunitu velmi vážně. Postupně se tak začaly objevovat změny v HW i souvisejícím SW, které stále více přibližují grafické čipy potřebám obecného paralelního počítání. Tomuto vývoji napomohlo i převzetí firmy ATI gigantem AMD z roku 2006. Jak AMD(ATI) tak NVIDIA navíc postupně vyvíjejí a přivádějí na trh hardwarová řešení, která jsou postavena na nejmodernějších GPU, avšak která jsou přímo určena pro obecné výpočty. O těchto produktech se krátce zmíníme ve zbytku této práce.

Jak jsme již naznačili, vývoj v oblasti GPGPU probíhá ohromujícím tempem. Nejmodernější grafické čipy a na nich postavené vývojové platformy podstatně mění způsob, jakým budou vytvářeny budoucí GPGPU aplikace. V poslední kapitole této práce se tedy krátce zmíníme o těchto trendech a technologiích. Zatímco předchozí uvedený přístup ke GPGPU si může doma vyzkoušet každý čtenář, následující moderní pojetí GPGPU vyžaduje nejnovější HW konkrétních výrobců, jak bude vysvětleno dále. Sám autor této práce neměl v době jejího psaní takový HW k dispozici, podrobnější popis a vlastní aplikace moderního GPGPU tak budou obsahem práce navazující.

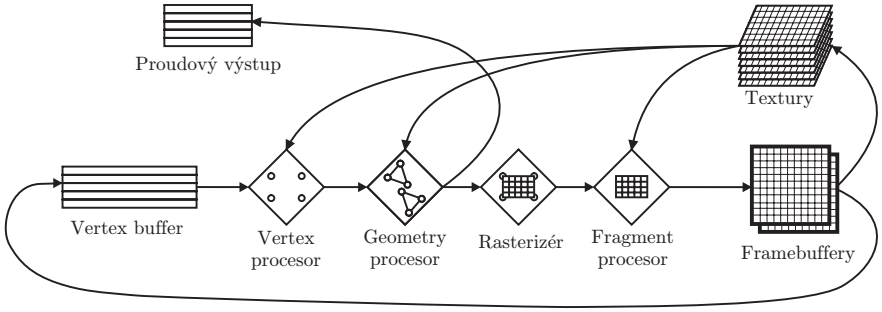
3.1 Moderní GPU a jejich přínos

V části 1.2.4 jsme vysvětlili, že ke klasifikaci GPU je vhodné použít MS DirectX API. Specifikace Direct3D 10 je součástí MS DirectX 10 (dále jen DX10) a znamená největší skok vpřed ve světě GPU od dob přechodu z pevné grafické pipeline na pipeline programovatelnou. DX10 přináší mnoho změn a vylepšení, po mnohých z nich navíc už delší dobu toužila GPGPU komunita. V následujícím textu proto nejdříve popíšeme novinky popsané v DX10, poté se budeme v několika odstavcích věnovat jedné z rodin GPU, které tuto specifikaci implementují, totiž procesorům NVIDIA GeForce řady 8.

3.1.1 To nejpodstatnější ze specifikace Direct3D 10

Nejprve poznamenejme, že změn v DX10 oproti předchozím verzím je poměrně hodně, nás však budou zajímat jenom ty nejpodstatnější a dále ty, které mají přímou souvislost s GPGPU a naší problematikou. Jedním z nejbolestnějších faktorů, který podstatně ovlivňuje výkon výsledných aplikací (nyní máme na mysli „nativní“ aplikace grafické stejně jako některé GPGPU aplikace) je potřeba časté komunikace mezi GPU a CPU. V grafické pipeline byly proto provedeny podstatné změny, které se snaží takovou komunikaci minimalizovat. Na obrázku 3.1 nalezne čtenář tradiční podobu grafické pipeline s novými prvky z DX10 (srovnej s obrázkem 1.4).

[1] z angl. High Performance Computing, jedná se o známou zkratku ze světa superpočítačů a paralelních výpočtů.



Obrázek 3.1: Tradiční pohled na grafickou pipeline v rámci DX10

V ní se nově vyskytuje tzv. *geometry procesor*[2] (dále jen GP). Ten podobně jako ostatní části programovatelné pipeline umožňuje spouštět uživatelské *geometry programy*, známější pod označením *geometry shadery*. GP dostává na vstup vrcholy jednoho grafického primitiva (bodu, úsečky nebo trojúhelníka), na výstupu potom generuje vrcholy žádného nebo několika grafických primitiv. GP tedy skutečně může vytvářet další geometrii, která v původní 3D scéně vůbec nebyla definována, stejně jako některé objekty „zahazovat". Jelikož GP má v době výpočtu k dispozici všechny vrcholy daného primitiva, může efektivně určovat například rovnici roviny trojúhelníka a další geometrické veličiny.

Další důležitou novinkou v grafické pipeline je tzv. Proudový výstup[3]. Jeho zavedení znamená další krok k novému pojetí grafické pipeline. Ta byla dříve chápána jakoby „jednosměrně" - na začátek grafické pipeline vstoupí geometrická data, na jejím konci vystoupí rastrový obraz, poté znovu zasáhne CPU a pošle do pipeline nová data. Moderní GPU však umožňují techniky jako je render-to-texture nebo render-to-vertex-array a mnohé další, které do modelu grafické pipeline vnášejí nové datové proudy (viz. obrázek 3.1). Mechanizmus proudového výstupu tak umožňuje výstup GP přesměrovat jako proud vrcholů zpět na začátek grafické pipeline (případně načíst zpět do CPU). Proud vrcholů na výstupu GP tak může „postupovat" do rasterizační části pipeline, do proudového výstupu, nebo oběma směry současně.

Se změnami v grafické pipeline souvisí i změny paměťového modelu. Víme, že GPU ve své systémové paměti uchovává mnoho různých struktur, jako jsou textury, renderovací cíle, vertex buffery, buffery hloubky a mnohé další. DX10 všechny takové struktury označuje jako *zdroje*[4] (zdrojem je zkrátka oblast paměti, která může být připojena ke grafické pipeline, např. jako textura nebo jako renderovací cíl) a přináší jednotný způsob práce s nimi. Co je důležitější pro GPGPU, textury a renderovací cíle mohou být uspořádány do lineárního *pole zdrojů* (může obsahovat až 512 elementů) a připojeny k pipeline jako textura nebo renderovací cíl. Instrukce pro programování shaderů jsou potom rozšířeny tak, aby například FP mohl přistoupit k libovolné textuře takového pole textur. Tím odpadá problém s multitexturingem a počtem současně adresovatelných textur, který byl dosud omezen malou konstantou (tento problém jsme diskutovali v části 2.1.2).

Datové typy a shader model 4.0

Procosory na GPU vždy interně pracují s 32-bitovými daty. Ty mohou být interpretována jako čísla s plovoucí řádovou čárkou, nově však také jako celočíselná hodnota typu *integer*. Instrukční sada programovatelných procesorů byla zároveň rozšířena o celočíselné instrukce. Zavedení celočíselných

[2]Aby udržel konzistenci s ostatními částmi textu i citovanou literaturou, rozhodl se autor neprovádět úplný překlad tohoto pojmu, ačkoliv příhodnější označení v češtině by jistě mohlo znít *procesor geometrie*.

[3]z angl. Stream Output

[4]z angl. *resources*

datových typů odstraňuje problémy s adresováním popsané v části 1.4.2 a umožňuje také zcela nové GPGPU aplikace. Dodejme ještě, že v paměti GPU mohou být uloženy celočíselné hodnoty šířky 8,16 nebo 32 bitů typu *signed* nebo *unsigned* (VP, FP i GP však interně vždy počítají s daty šířky 32 bitů).

Podstatných změn doznaly také programovatelné procesory. Starší specifikace DirectX popisovaly VP a FP odděleně jako dvě různé programovatelné jednotky s do značné míry odlišnými vlastnostmi. DX10 přináší nový shader model 4.0 a jeho unifikovaný model. VP, FP i GP jsou nyní popisovány jediným virtuálním strojem se společnými vlastnostmi, stejnou množinou instrukcí, stejnými požadavky na přesnost výpočtů, stejnými počty vstupních, konstantních a pracovních registrů, stejnými možnostmi přístupu k texturám atp. Jednotlivé limity na počty zdrojů byly přitom podstatně rozšířeny a neměly by už pro vývojáře znamenat podstatné omezení, čtenář nechť sám srovná následující tabulku s tabulkou 1.2. V dalším textu uvidíme, že popsaná jednotná specifikace progra-

	DX10 SM 4.0
Počet instrukcí	$\geq$ 64K
Konstantní registry	16×4096
Pracovní (TEMP) registry	4096
Vstupní registry	16
Renderovací cíle	8
Maximální velikost 2D textury	8192×8192
Počet textur	128
Řízení toku programu	Dynamické

Tabulka 3.1: Vlastnosti shader modelu 4.0

movatelných procesorů skutečně odpovídá tomu, jak jsou tyto implementovány v HW.

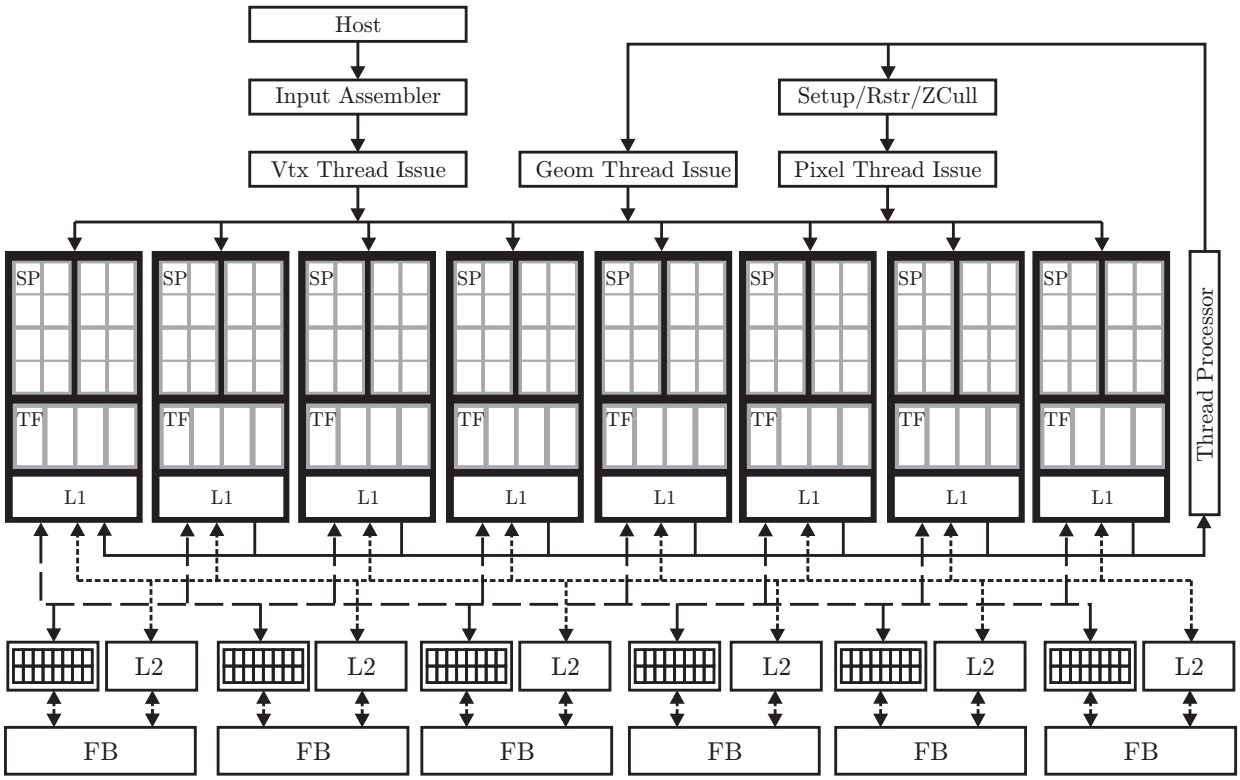
Změn které přináší DX10 je ve skutečnosti ještě mnohem více. Její autoři se spolu s výrobci HW snažili odstranit celou řadu omezení, která limitovala předchozí generace GPU v jejich přirozeném určení, totiž renderování 3D grafiky. Časté změny stavových veličin grafické pipeline, velké rozdíly mezi jednotlivými GPU, častá potřeba synchronizace CPU $\longleftrightarrow$ GPU, podstatná omezení na počty zdrojů (počty textur, instrukcí, ...), nejednotné požadavky na přesnost výpočtů, to jsou jen hlavní z problémů a omezení, která se DX10 snaží odstraňovat. Další informace nalezne laskavý čtenář v [22].

3.1.2 Architektura NVIDIA GeForce 8800

Dříve než si v poslední části tohoto textu řekneme něco o skutečném moderním GPGPU pojetí, zastavme se krátce nad jednou z prvních GPU implementujících DX10, tou je NVIDIA GeForce 8800 GTX. Povíme si jak jsou jednotlivé části z DX10 implementovány v HW a co to znamená pro GPGPU. Jak jsme si řekli v předchozích odstavcích, původní grafická pipeline byla skutečně chápána jako „potrubí“ s jedním vstupem a jedním výstupem, byla sekvenčním spojením několika nezávislých částí (její velmi hrubou podobu jsme uvedli na obrázku 1.2) a takto byla také v HW implementována[5]. DX10 a jeho unifikovaný shader model 4.0 klade shodné požadavky na množiny instrukcí, přesnost výpočtů, počty zdrojů a mnoho dalších pro všechny typy shaderů. Autoři GeForce 8800 však tuto unifikaci dotáhli do krajnosti a vytvořili architekturu, postavenou výhradně okolo programovatelných procesorů, tzv. *stream procesorů* (dále jen SP).

Blokové schéma architektury GeForce 8800 je naznačeno na obrázku 3.2. Implementace grafické pipeline se podstatně změnila, ta je nyní do značné míry cyklicky orientována. SP zpracují vstupní vrcholy (aplikací vertex shaderu) a výstup je poté odeslán zpět na vstup SP pro aplikaci geometry

[5]Skutečná HW implementace grafické pipeline je samozřejmě mnohem jemnější, každá z hlavních fází (např zpracování fragmentů) se skládá z desítek nebo stovek sekvenčních kroků s unikátní HW implementací.



Obrázek 3.2: Blokové schéma GeForce 8800 GTX

shaderu, situace se dále znovu opakuje ve fázi zpracování fragmentů. SP zastávají činnost všech typů programovatelných procesorů. Takový přístup umožňuje efektivně vytěžovat dostupné prostředky. Jak jsme si řekli v předcházejícím textu, starší GPU mají v HW implementován pevný počet VP i FP. Aplikace požadující velmi intenzivní zpracovávání vrcholů pak například nevytěžuje dostupné FP, které ke zpracovávání vrcholů nelze použít. Architektura GeForce 8800 však takové problémy odstraňuje a umožňuje efektivně přidělovat SP těm druhům výpočtů, které jsou v daném okamžiku třeba. Každý výpočet na GPU probíhá ve vlákně, těch dokáže GeForce 8800 zpracovávat tisíce současně, přičemž GPU sama efektivně plánuje, kterým vláknům (může se jednat o vrcholy, geometrii, fragmenty, nebo obecné výpočty) přidělí v daném okamžiku prostředky.

Popisovaná (nejvýkonější) varianta GeForce 8800 GTX má v HW implementováno 128 stream procesorů které jsou sdruženy do skupin a sdílejí jednotky pro adresování a filtrace textur a L1 cache (jak je patrné z obrázku 3.2). Blok osmi SP vždy tvoří tzv. *multiprocesor* a chová se jako SIMD paralelní zařízení. SP jsou navíc nově koncipovány jako *skalární* procesory (srovnej s popisem architektury v části 1.2.4). Autoři architektury GeForce 8800 totiž zjistili, že ve stále delších a složitějších shaderech se objevovalo stále více skalárních výpočtů, které šlo jen velmi obtížně mapovat efektivně na vektorový HW (víme, že předchozí verze VP a FP obsahovaly instrukce pro paralelní zpracování 4-složkových vektorů). Obrácený přístup přitom problematický není, nezávislé vektorové výpočty lze vždy namapovat na paralelní skalární architekturu, představovanou stream procesory. Důležitým vývojem prošla také vnitřní logika spouštění instrukcí. Při čtení dat z textury a případné filtraci (někdy může trvat i několik taktů hodin) se GPU snaží na daném SP zpracovávat další nezávislé matematické intrukce, může dokonce přepnout a spustit zpracování úplně jiného vlákna, jestliže vlákno původní čeká na data z textury. Poslední modifikace, kterou zmíníme v tomto odstavci, se týká větvení a dynamického řízení toku programu vůbec. Řekli jsme, že jednotlivé multiprocesory stále pracují SIMD

paralelním způsobem. V principu tak stále zůstává v platnosti většina omezení, popsaných v části 1.4.3, podstatně se však zmenšila velikost bloku fragmentů, zpracovávaných najednou SIMD způsobem. GeForce 8800 tak může měnit směr toku programu (beze ztráty efektivity) pro každých 32 fragmentů.

3.2 NVIDIA CUDA

Spolu s inovativním HW, který implementuje DX10 a o němž jsme si psali v předcházejících odstavcích, dali vývojáři z NVIDIE vzniknout ještě jedné velice důležité technologii. CUDA[6] představuje novou architekturu, která je kombinací SW a HW řešení a umožňuje využívat GPU jako obecný datově-paralelní výpočetní prostředek. Tato architektura znamená první skutečně propracovanou abstrakci, která umožňuje využívat GPU pro parallení výpočty bez znalosti grafického API a souvisejících principů. Zdaleka se přitom nejedná pouze o SW knihovnu, srovnatelnou s prostředky uvedenými v části 1.6. Některé části CUDA jsou implementovány v samotném HW a v ovladačích grafické karty, přičemž na nejvyšší úrovni samozřejmě stojí API a runtime knihovna. Ve světle architektur jako je CUDA označují již dnes někteří vývojáři postupy a techniky uvedené v kapitolách 1 a 2 jako „GPGPU ze staré školy". Až dosud se totiž GPGPU vývojáři museli podřizovat trhu počítačových her a mapovat své výpočty (často značně krkolomně) na HW, který pro takové výpočty nebyl původně určen a poskytoval pro ně jen omezené prostředky. CUDA a další technologie spolu s moderními GPU a souvisejícím HW však znamenají malou revoluci - vznikají a přizpůsobují se potřebám obecného počítání, v budoucnu lze navíc důvodně očekávat prohlubování tohoto trendu. Jak již bylo zdůrazněno, autor této práce neměl v době jejího psaní relevantní zkušenost s touto architekturou a s HW, který by ji podporoval, proto závěrečné odstavce tohoto textu představují jen teoretické minimum a měly by se stát základem pro práci navazující. Naprostá většina následujícího textu bude vycházet z [23].

Z pohledu vývojáře se SW část CUDA skládá z ovladače (je třeba nainstalovat jako ovladač grafické karty), runtime knihovny (a souvisejícího API) a knihoven vyšší úrovně. Prostředí využívá multitasking operačního systému a umožňuje běh několika CUDA a grafických aplikací zároveň.

Programovací model a jeho implementace v HW

CUDA považuje obsluhovaný HW za *zařízení* (*device*) (zařízením tedy rozumíme HW prostředek, obsluhovaný CUDA architekturou, CPU řídící celý proces se v tomto kontextu nazývá *hostitelem* (*host*)), které dokáže paralelně zpracovávat velké množství *vláken* (*threads*). Funkce, provádějící intenzivní výpočty, které lze nezávisle paralelizovat, mohou být kompilovány pro zařízení do podoby *jader* (*kernels*) a spouštěny na GPU právě v podobě mnoha vláken. Skupina vláken, spouštěných v rámci jednoho jádra, je rozdělena do *bloků* (*blocks*). Zde přichází jedna z nejpodstatnějších inovací architektury CUDA. Vlákna v bloku mohou navzájem komunikovat a sdílet společná data skrze rychlou sdílenou paměť, mohou také synchronizovat své výpočty (specifikací synchronizačních bodů v jádře). Každé vlákno je v rámci bloků označeno jednoznačným identifikátorem. Každý blok může být deklarován jako 1D, 2D nebo 3D pole vláken, příslušný identifikátor vlákna je potom jednosložkový, dvousložkový nebo třísložkový vektor. Jelikož počet vláken v bloku je omezen (v současnosti konstantou 512), mohou bloky stejného rozměru a dimenze tvořit tzv. *grid* (zde se autor záměrně vyhýbá překladu), který sdružuje všechna vlákna, spouštěná v rámci jednoho jádra. Vlákna v různých blocích nicméně nemohou navzájem komunikovat a synchronizovat svoji činnost. Samotný grid může představovat 1D nebo 2D pole bloků, jež jsou také označeny jedinečnými identifikátory.

S popsanou terminologií úzce souvisí paměťový model. Vlákna mají přístup do lokální paměti integrované na čipu, dále do DRAM paměti zařízení, tento paměťový prostor je však dále rozdělen na několik částí. Vlákna tak mohou provádět:

[6] z angl. Compute Unified Device Architecture

- čtení\zápis do registrů vlákna
- čtení\zápis do lokální paměti vlákna
- čtení\zápis do sdílené paměti bloku
- čtení\zápis do globální paměti gridu
- čtení z konstantní paměti gridu
- čtení z paměti textur gridu

Přitom lokální a globální paměť jsou implementovány jako oblasti v systémové paměti zařízení umožňující čtení i zápis, konstantní paměť a paměť textur jsou také umístěny v systémové paměti, ale umožňují pouze čtení.

Samotné zařízení je v HW implementováno jako skupina *multiprocesorů*, přičemž vlastní multiprocesor představuje SIMD architekturu (srovnej s popisem architektury GeForce 8800 v předcházející části). Každý multiprocesor má na čipu integrovány čtyři druhy rychlé paměti - množinu pracovních registrů pro každý procesor, paralelní datovou cache neboli sdílenou paměť (implementuje sdílenou paměť bloku) sdílenou všemi procesory, konstantní cache sdílenou všemi procesory (pro urychlování čtení z konstantní paměti gridu) a cache textur sdílenou všemi procesory (pro urychlení čtecích operací z paměti textur gridu). Každý multiprocesor přistupuje do paměti textur skrze *texturovací jednotku*, která umožňuje různé druhy adresování a filtrace textur (CUDA ve skutečnosti uvaluje některé restrikce na klasické chování textur známé z GPU, v současné době například umožňuje pouze bilineární interpolaci textur).

Každý grid vláken je na zařízení spouštěn tak, že jeden nebo více bloků je mapováno na jeden multiprocesor. Vlákna v bloku jsou rozděleny na tzv. *warps*, což jsou skupiny vláken stejné velikosti (v současné specifikaci obsahuje každý warp 32 vláken), splňující požadavky SIMD architektury. Každý warp tak může být spuštěn na SIMD multiprocesoru, přičemž *plánovač vláken* periodicky přepíná mezi jednotlivými warpy pro maximální využití výkonu multiprocesoru. Každý blok je vždy zpracováván jediným multiprocesorem, což umožňuje mapovat sdílený paměťový prostor do rychlé paměti integrované na čipu pro dosažení nejrychlejších možných paměťových přístupů. Základní představení programovacího modelu architektury CUDA a jeho implementace v HW byly bezpochyby velice stručné (další informace je možné nalézt v [23]), spíše než mnoho nových pojmů (majících často specifický význam v architektuře CUDA) by si tak měl pozorný čtenář z uvedeného textu odnést dva velice důležité poznatky:

1. CUDA umožňuje jistou formu komunikace a synchronizace mezi vlákny, skrze přístup do rychlé sdílené paměti. Jednotlivá vlákna tak mohou být navzájem závislá (srovnej s proudovým programovacím modelem uvedeným v části 1.2.3)
2. CUDA rozděluje DRAM zařízení do několika paměťových prostorů, co však je důležité, umožňuje do některých z těchto prostorů náhodný přístup pro zápis, tedy umožňuje *scattering* (srovnej s částí 1.4.2)

Způsob programování a API

Autoři CUDA se jako vždy vydali cestou nejmenšího odporu a vlastní programovací jazyk tak vzniknul minimálním rozšířením jazyka C. Toto rozšíření zahrnuje kvalifikátory pro určení typu funkce (rozlišuje, zda může být volána na hostiteli nebo na zařízení), kvalifikátory pro určení typu proměnné (určují v jakém paměťovém prostoru zařízení bude proměnná umístěna), novou direktivu pro určení způsobu spouštění jader na zařízení a několik předdefinovaných proměnných. Samotná runtime knihovna a související API jsou potom rozděleny na tři části podle toho, zda jsou dané funkce spouštěné na hostiteli, na zařízení, nebo mohou být mapovány na oba prostředky současně. O překlad zdrojových souborů s CUDA-rozšířenou syntaxí se stará proprietární překladač `nvcc`, jehož základním úkolem je oddělit kód určený pro spuštění na zařízení a přeložit jej do binární podoby. `nvcc` navíc přejímá filosofii obecných překladačů typu `gcc`, je schopen zpracovat celou řadu parametrů a umož-

ňuje volání dalších podřízených nástrojů a překladačů (např. překladač jazyka C, dostupný na dané platformě).

3.3 Shrnutí a předpovědi do budoucna

NVIDIA CUDA zdaleka není jediným dostupným prostředkem pro moderní GPGPU. Firma AMD(ATI) vyvinula vlastní technologii, většinou označovanou zkratkou CTM[7], která si klade za cíl zpřístupnit pole stream procesorů jejich HW pro vysoce paralelní výpočty bez použití grafického API. Na rozdíl od CUDA se však jedná o poměrně nízkoúrovňové řešení a my se jím v této práci nebudeme dále zabývat, případné zájemce odkazujeme například na [25]. Nejmodernější grafické karty a jejich unifikované architektury položily základ pro vznik dalšího GPGPU fenoménu, jejich výrobci si totiž uvědomili možnosti nově se otevírajícího trhu a začali vyvíjet zařízení určená pro HPC. Světlo světa tak v nedávné době spatřily NVIDIA Tesla a AMD Stream Processor. V obou případech se jedná o HW řešení opřená o GPU daných výrobců, upravená v několika ohledech (typicky chybí konektory pro výstup obrazu, liší se konfigurace a množství paměti, takty hodin a některé další vlastnosti) pro HPC. Více informací o těchto architekturách lze nalézt ve zdrojích v příloze A. V budoucnu lze navíc důvodně očekávat, že HW pro GPGPU bude představovat samostatnou vývojovou větev a bude stále lépe přizpůsoben požadavkům HPC. Objevily už se například hlasy, oznamující podporu fp64 (`double`) aritmetiky v budoucím HW, ačkoliv ta se možná bude týkat právě jen specializovaných odnoží HW, určených pro GPGPU.

[7] z angl. Close To Metal

Závěr

Tato práce měla postupné cíle. Nejdříve jsme se snažili s mnoha desítek článků a internetových zdrojů poskládat ucelený obraz toho, co je to vlastně GPGPU. Následovalo zkoumání detailních zákonitostí tohoto druhu numerického počítání a hledání vhodných problémů, které lze nyní nebo v budoucnu touto metodou řešit. Jelikož jsme v době psaní neměli k dispozici moderní a výkonný HW, který by snesl srovnání s technologickou špičkou (nebo který by v některých ohledech vůbec umožňoval nejmodernější GPGPU pojetí), soustředili jsme se v této práci na technologickou a implementační stránku problematiky. V době, kdy s touto prací začínal, neměl její autor velké praktické zkušenosti s počítačovou grafikou a neměl vůbec žádné zkušenosti s hardwarem akcelerovanou grafikou a souvisejícími API jako je OpenGL nebo Direct3D, přitom je třeba přiznat, že taková předchozí zkušenost by poměrně výrazně usnadnila a urychlila pochopení celé techniky. Tato práce by nicméně v tomto směru mohla budoucím začínajícím vývojářům „usnadnit cestu“.

V první kapitole jsme se věnovali technologickým základům GPU, stručně jsme připomenuli některé pojmy z počítačové grafiky a naznačili jsme možnost použití GPU pro obecné výpočty. Následoval demonstrační příklad, který měl čtenáři umožnit pochopení základních principů skutečné implementace. Nakonec jsme provedli detailní zkoumání vlastností GPU a popsali způsob jejich dalšího využití k efektivním výpočtům. Ve druhé kapitole jsme si popsali implementaci dvou „skutečných“ GPGPU aplikací, jednalo se o řešení rovnice vedení tepla a implementaci LU rozkladu husté matice. V obou aplikacích jsme se soustředili na takové techniky a postupy, které výrazně souvisejí s architekturou GPU a které jsou pro GPGPU v mnoha směrech „charakteristické“. V poslední kapitole jsme se zabývali popisem moderního HW a naznačili jsme směr, jakým se bude GPGPU ubírat v budoucnu. Ve světle faktů uvedených v této kapitole můžeme bez uzardění prohlásit, že se tato fascinující technika začíná stále výrazněji prosazovat ve světě výkonných numerických výpočtů.

Na tuto práci bychom chěli v nejbližší budoucnosti navázat ve dvou směrech. První předpokládá využití moderního a výkonného HW k testům výkonu a efektivity zde uvedených i dalších GPGPU aplikací. Druhý směr spočívá v hledání a implementaci složitějších (komplexnějších) problémů metodou GPGPU, zejména pak moderními metodami popsanými v kapitole 3. V současné době se nabízí zejména implementace vhodného řešiče soustavy lineárních algebraických rovnic s řídkými maticemi a jeho použití na řešení některých implicitních numerických schémat.

Přílohy

Příloha A

Internetové zdroje

Na tomto místě poskytneme čtenáři alespoň základní přehled internetových zdrojů. Následující text obsahuje komentované odkazy na důležité internetové zdroje přímo či nepřímo související s textem této práce i GPGPU jako takovým. Konkrétní články a materiály o GPGPU jsou uvedeny v seznamu použité literatury. Přehled v této příloze má obecnější charakter a obsahuje internetové adresy, jejichž platnost v budoucnu však není a nemůže být zaručena.

Základní informace o GPGPU

- Hlavní portál do světa GPGPU, obsahuje nepřeberné množství informací a odkazů na další zdroje
  `www.gpgpu.org`

Grafické API a související zdroje

- OpenGL
  `www.opengl.org`
- DirectX - vše podstatné o Direct3D i dalších součástech tohoto multimediálního API
  `msdn.microsoft.com/directx`
- GLUT (OpenGL Utility Toolkit) - jednoduchý okenní manažer vytvořený pro výuku OpenGL, vyhledávaný pro jednoduchost a dobrou přenositelnost
  `www.opengl.org/resources/libraries/glut`
- GLEW (OpenGL Extension Wrangler Library) - velmi užitečná knihovna pro identifikaci a inicializaci rozšíření OpenGL
  `glew.sourceforge.net`
- Přehled OpenGL rozšíření
  `oss.sgi.com/projects/ogl-sample/registry/`

Stránky s výukovým materiálem

- Skvělé výukové stránky pokrývající základní GPGPU techniky od Dominika Göddekeho (naprosto zásadní informace pro začínající GPGPU vývojáře)
  `www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/index.html`
- Výuka GLSL
  `www.lighthouse3d.com/opengl/glsl/`

Jazyky vyšší úrovně pro programovaní shaderů

- Cg
  developer.nvidia.com/page/cg_main.html
- HLSL
  msdn2.microsoft.com/en-us/library/bb509561.aspx
- GLSL
  www.opengl.org/documentation/glsl/

Jazyky a knihovny nejvyšší úrovně

- Brook - programovací jazyk implementující proudový programovací model
  graphics.stanford.edu/projects/brookgpu/
- Sh - metaprogramovací jazyk pro programování shaderů
  www.libsh.org/

Ladící nástroje

- gDEBugger - oblíbený nástroj pro ladění OpenGL programů
  www.gremedy.com
- The Image Debugger - jednoduchý ladící nástroj založený na principu grafických výstupů
  billbaxter.com/projects/imdebug/
- Microsoft Shader Debugger - ladící nástroj integrovaný do MS Visual Studio IDE
  msdn.microsoft.com/archive/default.asp?url=/archive/en-us/directx9_c_summer_03/directx/graphics/tools/shaderdebugger.asp
- glslDevil - velmi zajímavý ladící nástroj, umožňující ladění shaderů napsaných v GLSL
  www.vis.uni-stuttgart.de/glsldevil/

Moderní GPGPU

- CUDA - moderní technologie pro GPGPU vyvíjená firmou NVIDIA
  developer.nvidia.com/object/cuda.html
- NVIDIA TESLA - řešení pro HPC firmy NVIDIA
  www.nvidia.com/object/tesla_computing_solutions.html
- AMD Stream Processor - řešení pro HPC firmy AMD
  www.nvidia.com/object/tesla_computing_solutions.html

Příloha B

Použité zkratky

V následující tabulce je pro snadnou orientaci čtenáře uveden stručný přehled většiny použitých zkratek. Některé pojmy se autor rozhodl nepřekládat a ponechat uvedené pouze jejich původní znění.

Zkratka	**Pojem**	**Český překlad (popis)**
Cg	C for graphics	(jazyk pro programování shaderů)
CPU	Central Processing Unit	
CTM	Close To Metal	(technologie firmy ATI pro provádění obecných paralelních výpočtů)
CTT	Copy To Texture	kopírování do textury
CUDA	Compute Unified Device Architecture	(unifikovaná architektura pro provádění obecných paralelních výpočtů na GPU)
DX*n*	DirectX n	(multimediální programové rozhraní firmy Microsoft - n je číslo uvedené verze)
FBO	Framebuffer Object	framebuffer objekt
FP	Fragment Processor	fragment procesor
GF*n*	GeForce 6	(architektura grafických karet NVIDIA GeForce řady n)
GLSL	OpenGL Shading Language	(jazyk pro programování shaderů)
GP	Geometry Processor	procesor geometrie
GPGPU	General-Purpose computation on Graphics Processing Units	provádění obecných výpočtů prostřednictvím grafického procesoru
GPU	Graphics Processing Unit	grafický procesor
HPC	High Performance Computing	provádění výpočtů na velmi výkonných počítačích (superpočítačích)
MIMD	Simple Instruction Multiple Data	různá instrukce aplikovaná na různá data (typ paralelního zpracování)
MRT	Multiple Render Targets	více renderovacích cílů
ODE	Ordinary Differential Equation	obyčejná diferenciální rovnice
PDE	Partial Differential Equation	parciální diferenciální rovnice
RTT	Render To Texture	renderování do textury
SIMD	Simple Instruction Multiple Data	stejná instrukce aplikovaná na různá data (typ paralelního zpracování)
SM*n*	Shader Model n	(součást specifikace Direct3D, pojednávající o programování shaderů)
SP	Stream Processor	stream procesor
VP	Vertex Processor	vertex procesor

Příloha C

Zdrojové kódy

Součástí této práce jsou zdrojové kódy uvedených GPGPU aplikací, jejich seznam nalezne čtenář v následujícím přehledu:

gpgpu
- `tutor_conv` ..... konvoluce - GPGPU demonstrační příklad
- `tutor_conv_brook` ..... konvoluce na GPU s pužitím knihovny Brook
- `heat_equation` ..... rovnice vedení tepla
- `lu_decomp` ..... LU dekompozice husté matice

Literatura

[1] Matt Pharr (Ed.). *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Addison-Wesley, 2005.

[2] Randima Fernando, Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wealey, 2003.

[3] John D. Owens et al. *A Survey of General-Purpose Computation on Graphics Hardware.* Computers Graphics Forum 26(1):80-113, 2007.

[4] Martin Rumpf, Robert Strzodka. *Graphics Processor Units: New Prospects for Parallel Computing.* Numerical Solution of Partial Differential Equations on Parallel Computers, Springer, 2005.

[5] OpenGL Architecture Review Board, Dave Schreiner, Mason Woo, Jackie Neider, Tom Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL.* Addison-Wesley, version 2(5th edition), 2005.

[6] OpenGL Architecture Review Board, Dave Schreiner. *OpenGL Reference Manual: The Official Reference Document to OpenGL.* Addison-Wesley, version 1.4(4th edition), 2004.

[7] Ujval J. Kapasi, Scott Rixner, Wiliam J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, John D. Owens. *Programmable Stream Processors.* IEEE Computer, pp. 54-62, 2003.

[8] Mark J. Kilgard, All About OpenGL Extensions, `http://www.opengl.org/resources/features/OGLextensions`

[9] Simon Green. *The OpenGL Framebuffer Object Extension.* Game Developers Conference, 2005.

[10] Randi J. Rost. *OpenGL Shading Language*, Addison Wesley, 2nd edition, 2006.

[11] NVIDIA company. *Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview*, `http://www.nvidia.com`, November 2006.

[12] Mark Harris, David Luebke. *Supercomputing 2006 Tutorial on GPGPU*, 2006.

[13] Jeff Bolz, Ian Farmer, Eitan Grinspun, Peter Schröder. *Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid.* ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003) 22(3):917-924, 2003.

[14] Jens Krüger, Rüdiger Westermann. *Linear Algebra Operators for GPU Implementation of Numerical Algorithms.* ACM Transactions on Graphics 22(3):908-916, 2003.

[15] Aaron E. Lefohn, Joe M. Kniss, Charles D. Hansen, Ross T. Whitaker. *A Streaming Narrow-Band Algorithm: Interactive Computation and Visualization of Level Set Surfaces.* IEEE Transactions on Visualization and Computer Graphics 10(4):422-433, 2004.

[16] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, Pat Hanrahan. *Brook for BPUs: Stream Computing on Graphics Hardware*. ACM Transactions on Graphics 23(3):777-786, 2004.

[17] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, Kevin Moule. *Shader algebra*. ACM Transactions on Graphics 23(3):787-795, 2004.

[18] Aaron E. Lefohn, Joe Kniss, Robert Strzodka, Shubhabrata Sengupta, John D. Owens. *Glift: An abstraction for generic, efficient GPU data structures*. ACM Transactions on Graphics 26(1):60-99, 2006.

[19] Nico Galoppo, Naga K. Govindaraju, Michael Henson, Dinesh Manocha. *LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware*. Proceedings of the ACM/IEEE Conference on Supercomputing, 2005.

[20] E. Vitásek. *Numerické metody*. SNTL, Praha, 1987.

[21] Gene H. Golub, Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, 3rd edition, 1996.

[22] David Blythe. *The Direct3D 10 System*. ACM Transactions on Graphics 25(3):724-734, 2006.

[23] NVIDIA company. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. `http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf`, 2007.

[24] Victor Podlozhnyuk. *Image Convolution with CUDA*. NVIDIA CUDA SDK, 2007.

[25] AMD company. *ATI CTM Guide - Technical Reference Manual*. `http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf`, 2006.