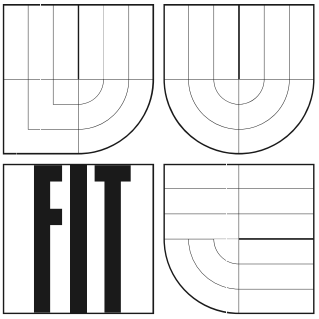
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Využití GPU pro všeobecné výpočty

Semestrální projekt

2006 Branislav Máček

Využití GPU pro všeobecné výpočty

Odevzdáno na Fakultě informačních technologií Vysokého učení technického v Brně, dne 3. ledna 2007.

Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením Ing. Igora Szökeho. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jméno Příjmení
Datum

Abstrakt

V súčasnosti grafické akcelerátory obsahujú programovateľné časti GPU – shadery, primárne určené na programovanie pokročilých grafických efektov (napr. soft shadows, HDR). GPU obsahujú desiatky až stovky paralelne usporiadaných výpočtových jednotiek. Tento paralelizmus ponúka vysoký výkon v aplikáciách s relatívne jednoduchými operáciami na obrovskom množstve dát.

Táto práca sa zaoberá popisom grafickej pipeline, približuje programovateľné časti GPU a ich programovací jazyk GLSL. Prezentuje princíp stream processingu a to na ukážke implementácie jednoduchého operátora lineárnej algebry pomocou OpenGL API a GLSL. Použitie výpočtov na GPU prisľubuje zrýchlenie trénovania neurónových sietí pre rozpoznávanie reči.

Klíčová slova

GPGPU, Stream processing, Shader program, OpenGL, GLSL, Linear Algebra

Abstract

Shaders are the most important part of graphic accelerators. In origin these programmable units were designed for computation of custom complex graphic effects. GPUs contain dozen to hundreds parallel processing units. This parallelism offer high performance in applications with relative simple operations on huge amount of data.

This work describes graphic pipeline, programmable parts of GPU and programming language GLSL. It presents principles of stream processing on example of simple linear algebra operator using OpenGL API and GLSL.

Keywords

GPGPU, Stream processing, Shader program, OpenGL, GLSL, Linear Algebra

Obsah

1 Úvod

Vďaka prudkému rozvoju trhu s počítačovými hrami existuje v súčasnosti vysoko výkonný hardvér, grafické karty, pre ktorý našli výskumníci aj iný spôsob ich využitia než len na zobrazovanie počítačovej grafiky. Túto oblasť využitia grafického hardvéru môžeme nazvať GPGPU - General-Purpose Computing on Graphics Processing Units.

Princíp výpočtov v tejto oblasti spočíva vo využití grafických procesorov GPU, ktoré obsahujú veľké množstvo paralelne usporiadaných výpočtových jednotiek – shaderov [Aga02]. Od svojho vzniku, prešli shadery v grafických akcelerátoroch obrovským vývojom. Od jednoduchých jednotiek na transformáciu vertexov, pre ktoré bolo treba písať programy v pseudo assembly jazyku navyše s obmedzeným počtom inštrukcií v programe, až po unifikované shadery s dostatočnými možnosťami na spustenie/vykonávanie veľmi zložitých programov.

Práca sa zaoberá zosumarizovaním možností grafických akcelerátorov na použitie pre všeobecné výpočty. Na začiatku je priblížená funkcionalita grafickej pipeline s dôrazom na jej programovateľné časti. V tretej kapitole je popísaný GLSL jazyk určený na programovanie shaderov. Následne sú vysvetlené princípy stream processingu a ukážka implementácie jednoduchého operátora lineárnej algebry touto technikou.

Negrafické počítanie na GPU má rôzne využitie: rýchle násobenie matíc, rýchle spočítanie FFT (Fast Fourier Transform) , Image/Video Processing a podobne. [Lars01]

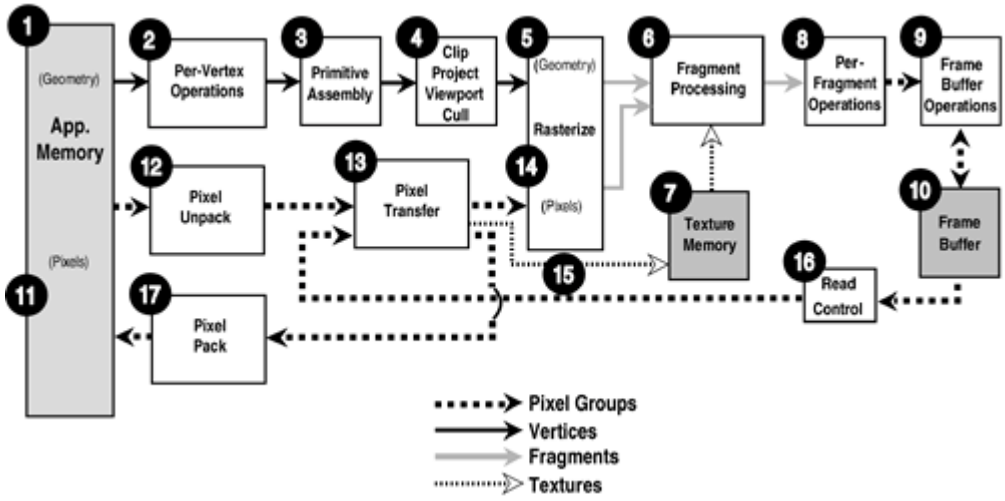
Táto práca nadväzuje na moju bakalársku prácu s témou programovania grafických efektov pomocou shaderov.

2 Architektúra grafických kariet

Táto kapitola popisuje pipeline grafických kariet. Je dôležitá na pochopenie súvislosti so Stream programovaním a ako je ho možné uskutočniť pomocou grafického API. Grafickú pipeline popíšem pomocou funkcii, ktoré poskytuje OpenGL API. Budem ho nazývať aj ako **fixná funkcionalita**. Na popis som vybral OpenGL pretože je to otvorený systém. Používa sa v profesionálnej počítačovej grafike napr. medicína. A taktiež pretože je to výukový nástroj na našej fakulte. Spôsob implementácie shaderov rozoberiem v nasledujúcej kapitole.

2.1 Grafická pipeline

OpenGL je API zamerané na zobrazovanie počítačovej grafiky. Model OpenGL sa dá charakterizovať ako klient-server. Aplikácia (klient) posiela príkazy implementácii OpenGL (serveru). Implementácia OpenGL je obyčajne súčasťou ovládača grafickej karty. Server príkazy interpretuje a vykonáva. Klient aj server môžu bežať na 2 rôznych počítačoch. Príkazy od klienta musia byť vykonávané v presnom poradí v akom sú posielané. OpenGL funguje ako stavový automat.



Obr. 2.1 Pipeline – fixná funkcionalita, zdroj[Ran04]

2.1.1 Vertex processing – spracovanie vertexov

Vertex processing je prvý stupeň OpenGL pipeline. Na obrázku je označený číslom (2). V tomto momente sú pozície vertexov transformované modelview a projekčnou maticou, normály sú inverzne presunuté podľa modelview matice. Textúrovacie súradnice sú presunuté podľa textúrovacích matíc. Základná farba je modifikovaná výsledkom z výpočtu svetla. Je aplikovaná farba materiálu a ďalšie.

Pretože najdôležitejšou časťou tohto stupňa sú transformovanie a osvetlenie, hovorí sa mu aj transformation and lighting (T&L). Aplikácia môže kontrolovať tento proces iba pomocou zásahov

do stavových hodnôt OpenGL. Pomocou *glEnable*/*glDisable* môže zapínať a vypínať svetlá, premennou *glLight* môže meniť ich parametre. Premennou *glMaterial* môže meniť vlastnosti materiálu. atď.

OpenGL ma špecifikovaný počet svetiel. Pomocou konštanty GL_MAX_LIGHTS možno zistiť aktuálny maximálny počet svetiel. Tento počet musí byt aspoň 8. Každé svetlo má niekoľko parametrov. Svetlo sa tak môže chovať ako priame, bodové alebo ako svetlo reflektoru. Taktiež možno nastavovať farbu svetla a jeho typ (difúzne, ambientné, zrkadlové). Tieto atribúty možno meniť pomocou funkcie *glLight*. Jednotlivé svetlá môžeme pomocou príkazov glEnable/glDisable zapínať a vypínať. Použitím týchto príkazov na symbolickú konštantu GL_LIGHTING kompletne zapneme alebo vypneme osvetlenie.

Počítať svetelné efekty samo o sebe nemá význam. Naším cieľom je osvetliť objekty, ktoré máme v scéne. Objekty v scéne majú tiež svoje svetelné vlastnosti – vlastnosti materiálu. Sú to farba svetla, ktoré emitujú, farba prostredia (ambient), rozptylu (diffuse) a zrkadlenia (specular), a lesk.(shininess). Vlastnosti materiálu sa dajú nastaviť zvlášť pre predný aj zadný povrch a menia sa funkciou *glMaterial*. Výsledné osvetlenie objektu je kombináciou typu jeho materiálu a svetelných podmienok v scéne. Funkciou glLightModel sa dajú nastaviť globálne parametre osvetlenia.

2.1.2 Primitive assembly – zloženie primitív

Po spracovaní vertexov, sú už všetky atribúty vertexov úplne stanovené. Dáta vertexov sú teda poslané do ďalšej časti pipeline – primitive assembly (3). Spolu s príkazom glBegin alebo s vertex poľom je prenesený atribút, ktorý určuje z koľkých vertexov sa skladá nasledujúca primitíva. Bod je jeden vertex, čiaru tvoria dva vertexy, trojuholníky vyžadujú 3 vertexy, štvoruholníky 4 vertexy a všeobecné polygóny ľubovoľný počet vertexov. Primitive assembly najskôr nazhromaždí potrebný počet vertexov a potom ich pošle do ďalšej časti pipeline. Táto časť pipeline je potrebná pretože nasledujúca časť vykonáva operácie nad množinami vertexov. Tieto operácie sú závislé na type konkrétnej primitívy.

2.1.3 Primitive processing – spracovanie primitív

Nasledujúca časť pipeline (4) pozostáva z viacerých rôznych častí, ktoré boli skombinované do jedného bloku. Prvý krok, ktorý nastáva je orezanie (clipping). Táto operácia porovnáva každú primitívu s užívateľom definovanými orezávacími rovinami (clipping planes). Užívateľ tieto roviny nastavuje pomocou *glClipPlane* alebo pomocou (view volume). View volume sa nastavuje pomocou modelview a projekčnej matice. Ak je primitíva podľa týchto podmienok v zobrazovacom pohľade, je poslaná na ďalšie spracovanie. Ak úplne mimo pohľad, primitíva sa ďalej nespracúva. Ak je primitíva s časti v pohľade, je rozdelená (clipped) tak, že na ďalšie spracovanie sa pošle iba orezaná časť splňujúca podmienky zobrazovacieho pohľadu.

Ďalšia operácia, ktorá nastáva v tejto časti pipeline je premietnutie podľa perspektívy (perspective projection). Ak je tento pohľad aktívny, budú súradnice *xyz* každého vertexu upravené – predelené homogénnou súradnicou *w*. Každý vertex bude transformovaný podľa súčasného zobrazovacieho poľa (viewport transformation) a pre každý vertex budú vytvorené súradnice v okne obrazovky. Viewport sa nastavuje pomocou funkcií *glDepthRange* a *glViewport*.
V tejto fáze môže byť spravené ešte jedno orezanie. Pomocou *glEnable* a *glCullFace* sa dá zapnúť operácia, ktorá s použitím vypočítanej súradnice otestuje každý polygón, či je nasmerovaný k pozorovateľovi alebo od neho. Na základe tejto informácie možno vybrať, či sa majú orezať privrátené, odvrátené alebo oboje polygóny.

2.1.4 Rasterization – rasterizácia

Rasterizácia (5) je proces dekompozície primitív na malé kúsky zodpovedajúce pixelom v cieľovom frame buferi. Tieto kúsky sa nazývajú fragmenty. Čiara (definovaná dvomi vertexami) môže pokrývať na obrazovke napr. 10 pixelov. Proces rasterizácie konvertuje túto čiaru na 10 fragmentov. Fragment je dátová štruktúra. Skladá sa zo súradnice na obrazovke, hĺbky, ďalej obsahuje atribúty ako sú farba, textúrovacie súradnice atp. Hodnoty atribútov sú počítané interpoláciou z údajov vertexov. V momente rasterizácie majú vertexy primárnu farbu a sekundárnu farbu. Zavolaním funkcie *glShadeModel* môžeme určiť akým spôsobom budú farby použité. Buď budú interpolované (Smooth shading), alebo sa použije farba posledného vertexu pre celú primitívu (Flat shading).

Pre každú primitívu existujú iné pravidlá rasterizácie a iný stav OpenGL. Šírka bodov sa kontroluje pomocou *glPointSize*, ostatné parametre rasterizácie bodov možno nastaviť pomocou funkcie *glPointParameter*. Šírka čiar sa stanovuje funkciou *glLineWidth*, vyplňujúci patern sa nastavuje funkciou *glLineStipple*. Vyplňujúci vzor polygónov určuje funkcia *glPolygonStipple*. Polygóny možno vykresľovať vyplnené, alebo ako obrys, alebo ako body. Na to slúži funkcia *glPolygonMode*. Funkcia *glPolygonOffset* nastavuje stav, ktorý sa používa na výpočet hodnoty, ktorá môže zmeniť hodnotu hĺbky fragmentu. Funkcia *glFrontFace* nastavuje orientáciu polygónov, ktoré majú byť považované ako otočené k pozorovateľovi. Vyhladzovanie zúbkov, ktoré sa objavujú na primitívach po rasterizácii sa nazýva antialiasing a môže byť zapnutý volaním *glEnable* (GL_POINT_SMOOTH, GL_LINE_SMOOTH, alebo GL_POLYGON_SMOOTH).

2.1.5 Fragment processing

Po rasterizácii nasleduje niekoľko operácii, ktoré spoločne nazývame fragment processing (6). Asi najdôležitejšou operáciou, ktorá tu nastáva je textúrovanie. Počas tejto operácie sú súradnice textúr priradené k fragmentom použité na prístup do oblasti grafickej pamäte nazývanej textúrovacia pamäť – texture memory (7). Problematika textúrovania je dosť zložitá. V OpenGL existuje mnoho

mechanizmov ako pracovať s textúrami. Na operácie s textúrami bolo vytvorených mnoho rozšírení (extensions).

Medzi ďalšie operácie s fragmentami patrí aplikovanie hmly (fog). Je to modifikovanie farby fragmentu na základe jeho vzdialenosti od pozorovateľa. Ďalšou operáciou je sčítanie farieb (color sum). Color sum spočíva v kombinovaní hodnôt primárnej farby a sekundárnej farby fragmentu. Parametre hmly sa nastavujú funkciou *glFog* . Sekundárne farby sú atribúty vertexu a môžu byť predané pomocou funkcie *glSecondaryColor* alebo vypočítané v časti T&L.

Na konci jednotky pre fragment processing, sú fragmenty predané množine jednoduchých operácií nazývané per-fragment operations (8). Sú to tieto:

Pixel ownership test – určuje či je cieľový pixel viditeľný alebo schovaný za prekrývajúcim oknom

Scissor test – fragmenty sú orezané podľa obdĺžnikového regiónu vytvorenom funkciou *glScissor*

Alpha test – používa alpha hodnotu fragmentu a funkciu *glAlphaFunc*, aby rozhodol či ho vyradiť

Stencil test – využíva funkcie *glStencilFunc* a *glStencilOp* na porovnanie hodnoty v stencil bufery s referenčnou hodnotou

Depth test – pomocou *glDepthFunc* porovnáva hĺbku prichádzajúceho fragmentu s fragmentom vo frame buferi.

Blending – zmieša farbu prichádzajúceho fragmentu s fragmentom vo frame buferi, využíva funkcie *glBlendFunc*, *glBlendColor*, a *glBlendEquation*.

Dithering – vykoná logické operácie s hodnotou fragmentu

Všetky tieto operácie dnes možno efektívne hardvérovo implementovať. Konceptuálne sú to jednoduché operácie a súčasný hardvér je schopný takto spracovať milióny pixelov za sekundu.

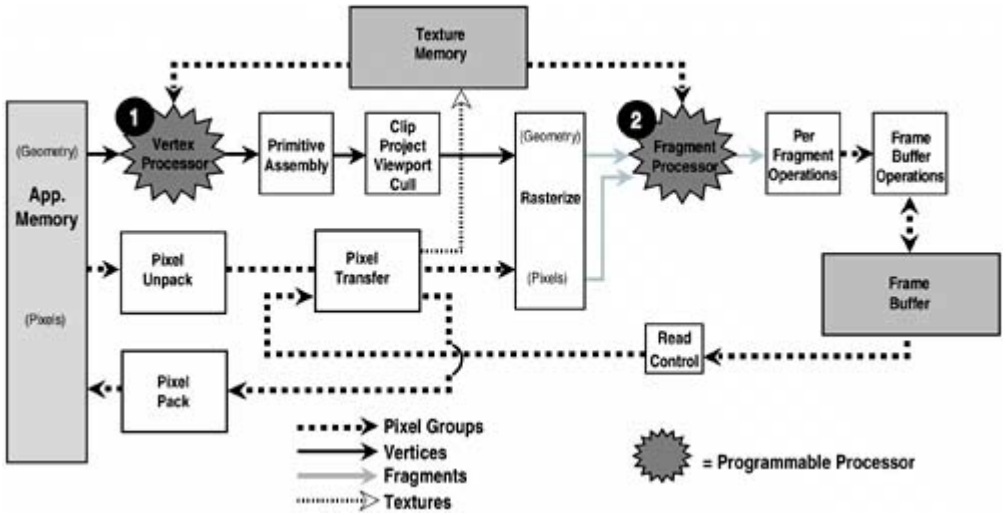
2.1.6 Frame buffer

Frame bufer uchováva hodnoty pixelov, ktoré budú vykreslené na obrazovku. Existuje veľmi veľa možností ako pracovať s frame buferom. To ako sa s frame buferom pracuje, možno kontrolovať OpenGL stavovými premennými. Hodnoty frame bufera sa inicializujú funkciou *glClear*. OpenGL podporuje zobrazovanie double-buffering. Frame bufer sa skladá z mnoha regiónov – buferov: front, back, left, right, atď. Funkciou *glDrawBuffer* možno vybrať, do ktorého z nich sa má renderovať. Existuje niekoľko schém buferovania a tým docieliť zvýšenie výkonu.

2.2 Programovateľnosť grafických kariet

Niektoré časti fixnej funkcionality, ktorú som popísal v minulej kapitole, nestačili požiadavkám na 3D zobrazovanie. Pridávanie nových efektov viedlo k neúmernému komplikovaniu hardvéru. Riešením bolo zavedenie programovateľných častí. Vhodné sa ukázalo nahradiť časti určené na spracovanie vertexov a fragmentov. Tieto programovateľné jednotky zdedili pomenovanie shadery. Podľa shaderov používaných v obore offline renderovania počítačovej 3D grafiky.

Práve tieto časti grafickej pipeline sú miestom, kde prebieha výpočet **kernel** funkcii **stream spracovania**. Súčasný vývoj naznačuje, že fixná funkcionalita bude v grafickom hardvéri ustupovať a programovateľné časti sa budú zväčšovať. Programovateľné časti narastajú nielen do veľkosti. Samé o sebe sa stávajú komplikovanejšie a teda schopné spracovať zložitejšie kernel funkcie.



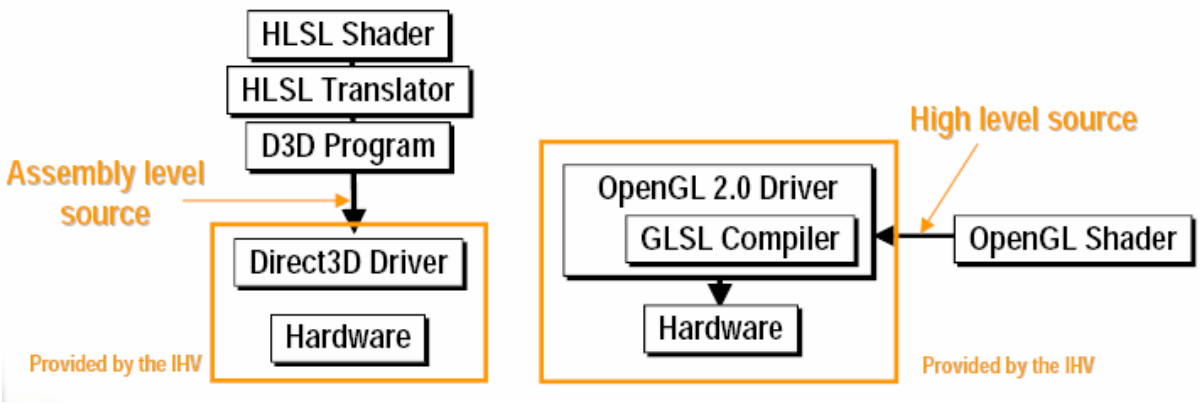
Obr. 2.2 Pipeline – umiestnenie shaderov, zdroj[Ran04]

Tento obrázok ukazuje umiestnenie vertex a fragment shadera v zobrazovacej pipeline (na obrázku označený ako vertex/fragment procesor). Samotný shader môže obsahovať niekoľko výpočtových jednotiek.

Spočiatku sa na programovanie shaderov používali priamo inštrukcie shader procesorov, teda assembler. Každý výrobca ale ponúka iné možnosti a programy napísané v strojovom kóde sú medzi rôznymi výrobcami nekompatibilné. Aj z tohto dôvodu vznikla požiadavka na high-level programovací jazyk. Inšpiráciou pre nový jazyk sa stal RenderMan a implementácia shading jazyka zo Standfordu. V roku 2002 boli v krátkom slede uverejnené jazyky Cg, HLSL a GLSL. Ich uvedeniu predchádzal niekoľkoročný vývoj. HLSL (high-level shading language) sa používa na platforme

DirectX. Jazyk bol vyvinutý v spolupráci firiem Nvidia a Microsoft. Jazyk Cg (C for graphics) je súčasťou SDK a vývojového prostredia shaderov CgFX firmy Nvidia. Firma Nvidia navrhla svoj jazyk aj pre OpenGL. Konzorcium ARB (Architecture Review Board) nakoniec vybralo pre OpenGL implementáciu od firmy 3DLabs - GLSL (OpenGL Shading Language).

HLSL a Cg sú veľmi podobné. Medzi aplikáciu a zobrazovacie API vkladajú interpretačnú vrstvu. Takéto riešenie má výhodu v štandardizovanom prístupe. Nevýhodou je zase oddelenie od hardvéru. Funkcionalita, ktorá nie je podporovaná medzivrstvou zostáva aplikácii skrytá. GLSL je úplne zakomponované do OpenGL. Je súčasťou API. Toto je výhoda, ktorá umožňuje vývojárom pristupovať k všetkým novým funkciám hardvéru (extensions). Jazyk Cg podporuje DirectX aj OpenGL. Pred kompilovaním treba zvoliť profil cieľovej platformy. Profily majú niekoľko úrovní rozdelených podľa schopností hardvéru (napr. shader model 1.1 alebo 2.0).



Obr. 2.3 Rozdiel medzi HLSL a GLSL

Všetky tieto programovacie jazyky vznikli približne v rovnakom čase. Jednotlivé firmy majú medzi sebou rôzne dohody a čo je dôležité bežia na takmer rovnakom hardvéri. Pravdepodobne preto majú ich jazyky podobnú syntax. Základom je jazyk C. Medzi ďalšie vlastnosti patria striktná typová kontrola, podpora vektorových a maticových operácií priamo v jazyku, podobným spôsobom komunikujú s hosťovským API, majú približne podobný zoznam vstavaných premenných a vstavaných funkcii. Nepodporujú ukazovatele. Keďže GLSL je súčasťou OpenGL, programátor môže zvnútra shader programu pohodlne pristupovať k stavovým premenným OpenGL. A to v podobe vstavaných (build-in) uniformov. Oproti tomu program napísaný v jazyku Cg sa dodáva aplikácii v strojovom kóde. Kôli tomu je komunikácia s OpenGL riešená formou symbolických premenných. Programátor musí všetky uniformy, ktoré chce používať, uviesť vo vstupnej štruktúre INPUT a výstupnej štruktúre OUT.

3 OpenGL Shading Language

GLSL sú v skutočnosti 2 úzko prepojené jazyky. Jeden určený pre vertex druhý pre fragment procesor. Majú rovnakú syntax. Líšia sa iba množinou vstavaných premenných a vstavaných funkcií. V tejto kapitole sa hlavne zameriam na prepojenie GLSL s OpenGL. Podrobný popis syntaxe a vstavaných funkcií je v príručke *The OpenGL Shading Language, Language version 1.10, Document revision 59*.

Terminológia:

vertex/fragment procesor – hardvér, na ktorom beží shader

vertex/fragment shader – výsledok kompilovania zdrojových súborov

shader program – výsledok linkovania vertex a fragment shaderov

Proces kompilovania a linkovania je podobný na aký sme zvyknutý z jazyka C. Jeden shader možno kompilovať z viacerých zdrojových súborov. Práve jeden zdrojový súbor musí obsahovať funkciu main. Súbory musia mať rovnaké globálne premenné.

3.1 Dátové kvalifikátory

Dátové kvalifikátory sú označenia premenných a bližšie špecifikujú ich vlastnosti. Sú to napríklad kvalifikátory **in** – premenná určená iba na čítanie, **out** – premenná určená iba na zápis, **inout** – určená na zápis aj na čítanie. Bližšie si predstavíme kvalifikátory **uniform**, **attribute** a **varying**. Kvalifikujú premenné určené na komunikáciu s OpenGL.

3.1.1 Uniform

Uniform je kvalifikátor používaný na deklarovanie globálnych premenných. Uniformy slúžia na komunikáciu s OpenGL a to v smere do shadera. Hodnota uniform premenných zostáva počas spracovania jednej primitívy nemenná. Do uniform premenných sa nedá zapisovať. Uniformom môže byť akýkoľvek dátový typ GLSL. GLSL obsahuje jednak vstavané (build-in) uniformy, tiež je možné deklarovať vlastné uniformy podľa potreby. Vstavané aj vlastné uniformy zdieľajú určitú implementačne závislú veľkosť pamäte. Deklarované ale nepoužité premenné sa do limitu nezapočítavajú. Hlavným znakom uniformu je že počas spracovania jednej primitívy, nemení hodnotu. Jeho hodnota obyčajne zostáva rovnaká počas spracovávania viacerých primitív. Najčastejšie sú to transformačné matice a podobne.

3.1.2 Attribute

Attribute sú globálne premenné prístupné iba vertex shaderu. Slúžia na komunikáciu z fixnou funkcionalitou OpenGL. Ich hlavným znakom je, že menia hodnotu s každým prichádzajúcim vertexom. Do attribute premennej nemožno zapisovať. Polia a štruktúry nemôžu byť atribútmi. Podobne ako uniformy aj atribúty môžu byť vstavané a užívateľom definované. Maximálne množstvo prenášaných atribútov je implementačne závislé.

3.1.3 Varying

Varying sú globálne premenné, ktoré sprostredkúvajú komunikáciu medzi vertex shaderom, fixnou funkcionalitou a fragment shaderom v smere do fragment shadera. Vertex shader môže tieto premenné čítať aj zapisovať, fragment shader iba čítať. Varying premenné sú počítané na základe vertexov, do fragment shadera sú prenášané hodnoty korektne interpolované podľa perspektívy. Je dôležité, aby typy a názvy varying premenných vo vertex a fragment shaderi navzájom zodpovedali, ináč sa shadery nepodarí zlinkovať. Fragment shader nemusí využiť všetky varying premenné deklarované vo vertex shaderi. Štruktúry nemožno deklarovať ako varying. Ak nie je aktívny fragment shader, vertex shader je zodpovedný za zápis varying premenných vyžadovaných fixnou funkcionalitou OpenGL. Ak nie je aktívny vertex shader, fixná funkcionalita vypočíta a zapíše vstavané varying premenné pre fragment shader.

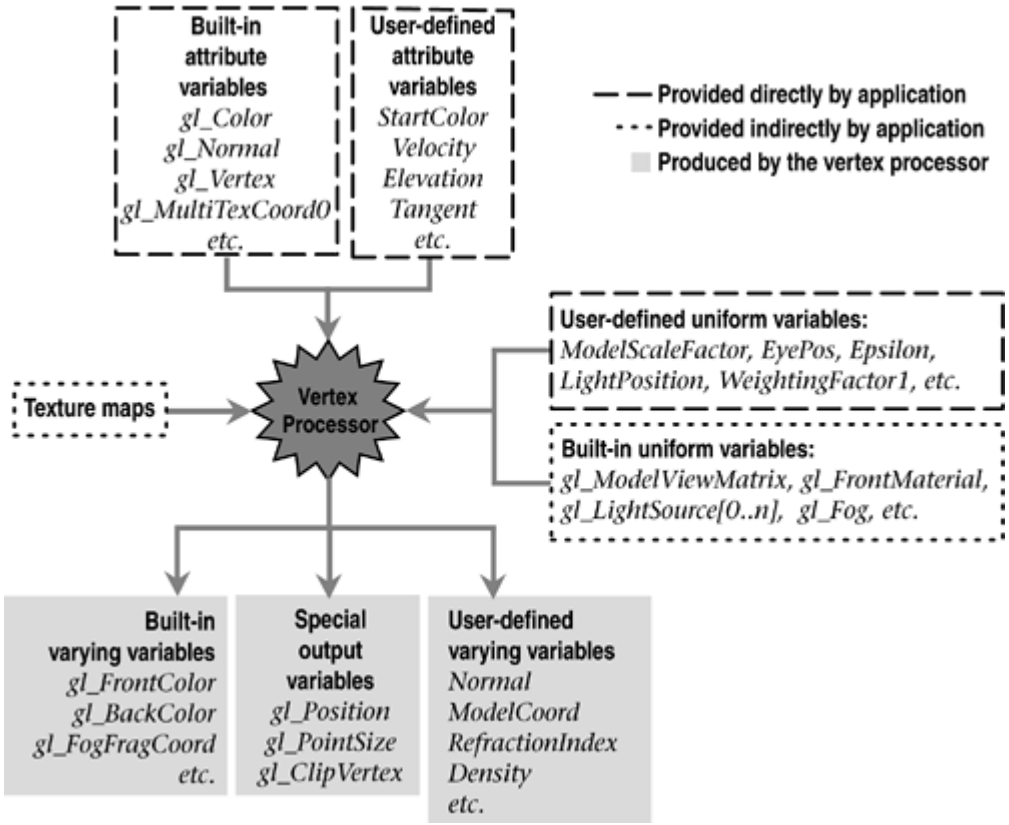
3.2 Vertex procesor

Je programovateľná jednotka, ktorá spracúva prichádzajúce vertexy a dáta k nim pripojené. Vertex procesor je určený vykonávať grafické operácie:

1. transformácie vertexov
2. generovanie a transformovanie koordinácii textúr
3. osvetlenie
4. aplikovanie farby na vertexy

Nasledovný obrázok ilustruje spôsob komunikácie vertex procesora s fixnou funkcionalitou OpenGL. Na vstupe sú nasledovné dáta. Dáta prenášané s každým vertexom pomocou attribute premenných. Medzi najčastejšie používané vstavané attribute premenné patria *gl_Color, gl_Normal, gl_Vertex* atď. Užívateľsky definované attribute premenné sa najčastejšie používajú na prenos dát napr. fyzikálnych veličín rýchlosť, zrýchlenie , čas a pod. Ďalej sú to uniform premenné. Medzi vstavané uniform premenné patria transformačné matice, premenné špecifikujúce parametre svetla a pod. Vertex procesor môže taktiež čítať z textúr.

Vertex procesor zapisuje do nasledovných výstupných premenných. Najdôležitejšia je ***gl_Position***. Je to povinná premenná, kam sa zapisuje spracovaný vertex a posiela ďalej do pipeline. V procese rasterizácie a orezania možno využiť premenné *gl_PointSize* a *gl_ClipVertex*. Vertex procesor môže zapisovať do niekoľkých vstavaných varying premenných. Keďže Fragment shader nepodporuje attribute premenné, užívateľský definované varying premenné sú jediný spôsob ako mu možno predať dáta meniace hodnotu s každým vertexom.



3.1 Schéma vstup/výstup vertex procesor, zdroj [Chan02]

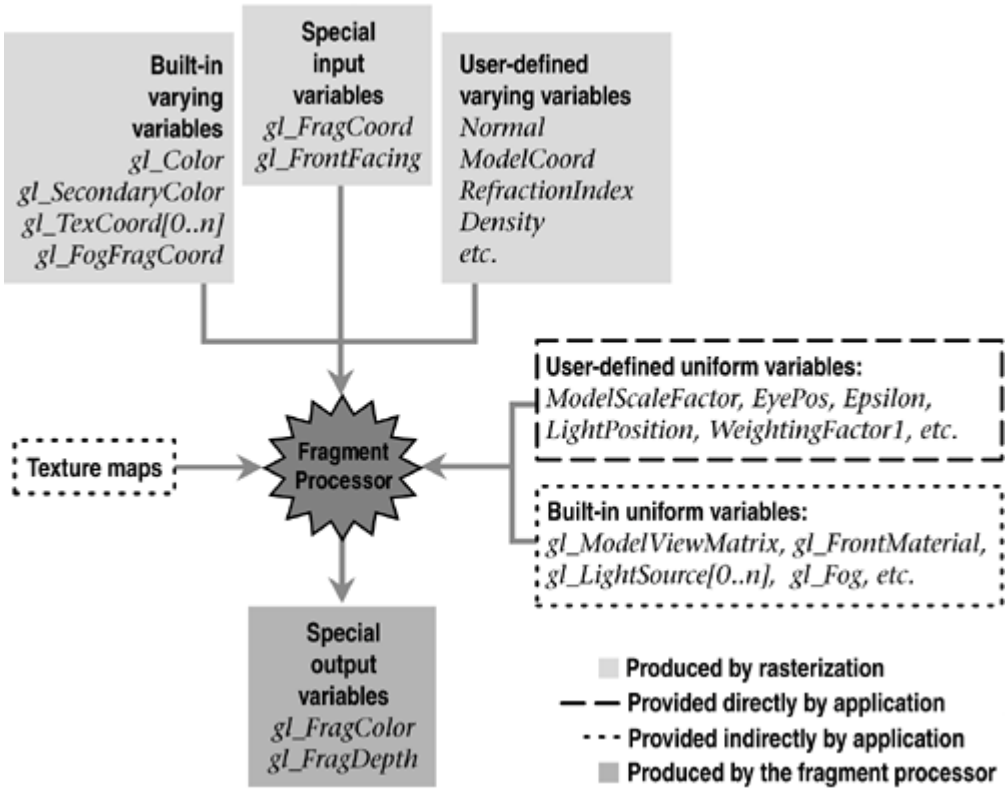
Ak je fragment shader aktívny, výstup vertex shadera musí súhlasiť z jeho vstupom. Ak fragment shader nie je aktívny, vertex shader musí uspokojiť potreby fixnej funkcionality.

3.3 Fragment procesor

Fragment procesor je programovateľná jednotka určená na paralelné spracovanie/farbenie plôch. Fragment shader je vykonaný pre každý fragment vyprodukovaný fixnou funkcionalitou v procese rasterizácie. Pri každom fragmente, má fragment shader prístup k interpolovaným hodnotám všetkých varying premenných: Color, normal, texture coordinates, arbitrary values.

Hodnoty vypočítané fixnou funkcionalitou medzi vertex procesorom a fragment procesorom sú prístupné pomocou špeciálnych premenných. Premennými *gl_FragCoord* a *gl_FrontFacing* sú

prenesené hodnoty pozície fragmentu v okne (window coordinate position) a či bol fragment bol generovaný rasteriziáciou čelnej primitívy.



3.2 Schéma vstup/výstup fragment procesor, zdroj[Chan02]

Tak ako vo vertex shaderi, aj vo fragment shaderi je možné pomocou uniformov pristupovať k OpenGL stavovým hodnotám (glColor a pod.). OpenGL stav je rovnako prístupný vo vertex shaderi aj vo fragment shaderi. To znamená, že pomocou fragment shadera možno prevádzať napríklad operáciu osvetlenia podobnú ako robí T&L. Fragment shader to ale dokáže per pixel, takže kvalitnejšie.

Veľkou výhodou fragment shadera je, že dokáže pristupovať neobmedzene krát do textúrovacej pamäte a načítané hodnoty ľubovoľne kombinovať. Fragment shader je schopný čítať viac hodnôt z jednej textúry (multiple values from a single texture), a tiež viac hodnôt z viacej textúr (multiple values from multiple textures). Výsledok prístupu k textúre môže byť základom pre čítanie v inej textúre (dependent texture read). Počet takýchto čítaní nie je obmedzený. Týmto spôsobom možno vo fragment shaderi implementovať algoritmus ray-tracing.

Fragment shader výsledok výpočtov ukladá do výstupných premenných *gl_FragColor* a *gl_FragDepth*. Fragment shader môže celý výpočet zahodiť (discard fragment). Výsledok fragment shadera je ďalej poslaný do zvyšku OpenGL pipeline. Fragmenty sú následne zapísané do frame bufera.

3.4 GLSL kompilátor a linker

Kapitola popisuje spôsob prípravy, prepojenia a spúšťania shaderov z prostredia OpenGL.

3.4.1 Základný prehľad

Firma 3Dlabs v spolupráci so Standfordskou univerzitou vyvinula kompilátor jazyka GLSL a uvoľnila ho na bezplatné používanie [Stan07]. Jej motiváciou je, aby všetci dodávatelia grafických kariet mali v svojej implementácii OpenGL rovnakú podporu jazyka GLSL (tak zvaný front-end). Na výrobcoch grafických kariet je ponechaný back-end teda kompilácia do strojového kódu. Počíta sa s tým, že každý výrobca si takpovediac ušije kompilátor na mieru a optimalizuje na svoj hardvér. Kompilátor a linker sú teda súčasťou ovládača grafickej karty.

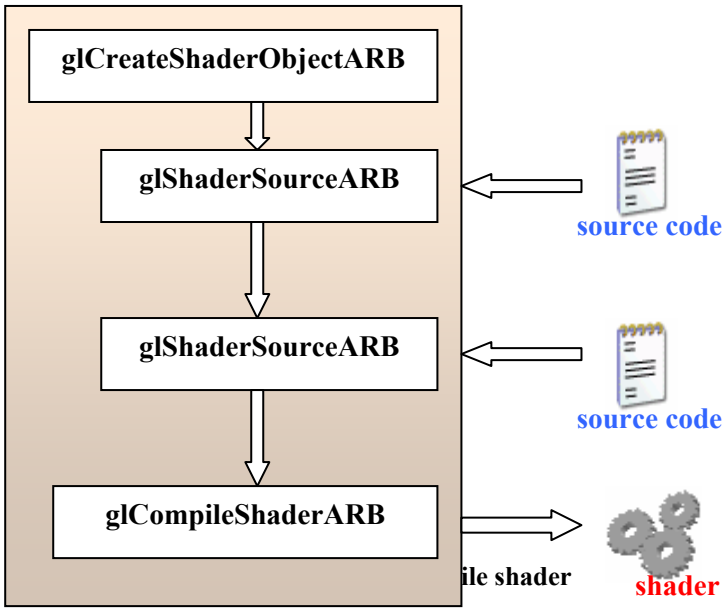
Priebeh kompilácie shaderu je podobný procesu kompilácie C programu. Jednotlivé zdrojové súbory zodpovedajú modulom. Výsledný shader možno zkompilovať z neobmedzeného množstva zdrojových súborov, ale iba jeden môže obsahovať funkciu *main()*. Shadery sú následne zlinkované do programu, ktorý je nakoniec nahratý do programovateľnej jednotky. Funkcie zabezpečujúce všetky operácie potrebné na kompiláciu GLSL súborov sú súčasťou OpenGL v.2.0.

3.4.2 Shader objekt

V tejto kapitole vysvetlím proces prípravy, prepojenia a vytvorenia shader objektu. Shader objekt je dátová štruktúra určená na identifikovanie shadera v prostredí OpenGL. Pre zostavovaný shader najskôr treba vytvoriť objekt, ktorý sa bude chovať ako kontajner. Na to slúži funkcia *glCreateShaderObject()*.

```
shaderID = glCreateShaderObjectARB(shaderType);
```

Parameter *shaderType* môže byť GL_VERTEX_SHADER_ARB alebo GL_FRAGMENT_SHADER _ARB. Každý shader, ktorý chceme zlinkovať musí mať svoj objekt.



K objektu pripojíme reťazec zdrojového kódu pomocou *glShaderSourceARB()*.

```
glShaderSourceARB(shaderID, numOfStrings, strings, lenOfStrings);
```

shaderID - handler shadera.

numOfStrings – počet reťazcov v poli.

strings – pole reťazcov

lenOfStrings – dĺžka každého reťazca, NULL znamená prázdny reťazec

Teraz môžeme shader zkompilovať.

```
glCompileShaderARB(shaderID);
```

Výsledok kompilácie možno skontrolovať pomocou funkcie

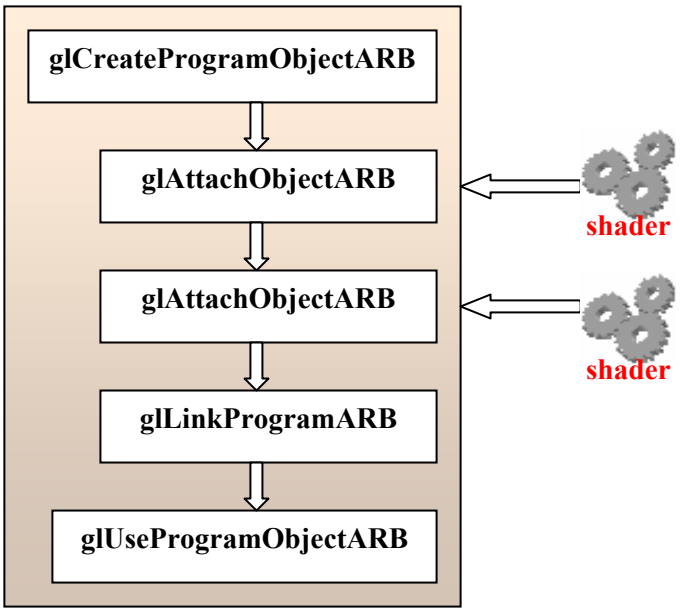
```
glGetObjectParameterfARB(shaderID, GL_OBJECT_COMPILE_STATUS_ARB, status);
```

3.4.3 Program objekt

Nasledujúci obrázok ukazuje potrebné kroky k vytvoreniu programu a jeho spusteniu v shader procesore.

Prvý krok spočíva vo vytvorení objektu, ktorý bude reagovať ako program kontajner. Slúži na to funkcia *glCreateProgramObjectARB*. Funkcia vytvorí handle na program kontajner.

```
GLhandleARB glCreateProgramObjectARB(void);
```



Obr. 3.4 Schéma - link program

Možno vytvoriť toľko programov koľko chceme. Počas renderovania sa môžeme medzi jednotlivými programami prepínať. Programy môžeme aj vypínať a nechať pracovať fixnú funkcionalitu. A to všetko je možné uskutočniť počas vykresľovania jediného snímku.

V nasledujúcom kroku pripojíme k programu vytvorené shadery. Táto fáza ešte nie je linkovanie. V tejto fáze len deklarujeme, že existujú také a onaké shadery a programy a že niektoré z nich chceme prepojiť. Nepotrebujeme ani zdrojové kódy shaderov. Jediné čo robíme je prepájanie programov a shader kontajnerov.

Na pripojenie shadera k programu slúži funkcia *glAttachObjectARB*

```
void glAttachObjectARB(GLhandleARB program, GLhandleARB shader);
```

Parametere:

program - handler na program.

shader - handler na shader ktorý chceme pripojiť

Posledný krok tvorby programu je linkovanie. V tomto momente už musíme mať zkompilované shadery.

```
void glLinkProgramARB(GLhandleARB program);
```

Takýmto spôsobom môžeme vytvoriť ľubovoľné množstvo programov. Funkciou *glUseProgramObjectARB*.sa môžeme medzi nimi prepínať.

```
void glUSeProgramObjectARB(GLhandleARB program);
```

Parameter program - handler programu, ktorý chceme aktivovať. Nula ak chceme zapnúť fixnú funkcionalitu

Po linkovaní programu je dovolené meniť zdrojový kód shaderov a prekompilovávať ich.

3.4.4 Prepojenie uniformov a atribútov

Aby malo používanie shaderov význam treba zabezpečiť komunikáciu medzi aplikáciou a shader programom. Na to slúžia uniformy a atribúty. Už sme si vysvetlili, že uniformy sú premenné, ktoré menia hodnotu maximálne raz za primitívu. Atribúty sú premenné, ktoré menia hodnotu s každým vertexom.

Postup prepojenia premenných aplikácie a shadera je nasledovný. Najskôr treba zistiť adresu uniformu alebo atribútu. Lenže tieto premenné sú v pamäti, až keď je shader program nahratý do procesora. Takže pred tým než budeme zisťovať adresu premennej, treba najskôr zlinkovať shader program. V niektorých implementáciách ho treba aj nahrať do shader procesora

Umiestnenie uniformu zistíme nasledovnou funkciou:

```
GLint glGetUniformLocationARB(GLhandleARB program, const char *name);
```

Ako parametre treba použiť handler shader programu a meno uniformu.

Návratová hodnota tejto funkcie je umiestnenie uniformu v pamäti. Na zapisovanie do tejto pamäte slúži niekoľko funkcií. Líšia sa počtom parametrov. Zapisujú hodnoty od float skalár až po 4 prvkove float vektory.

```
void glUniform1fARB(GLint location, GLfloat v0);
void glUniform4fARB(GLint location, GLfloat v0,
                    GLfloat v1, GLfloat v2, GLfloat v3);
```

Parametrami sú *location* získaná funkciou *glGetUniformLocationARB* a float hodnoty v0 až v3. Namiesto samostatných premenných možno predať pole:

```
GLint glUniform[1až4]fvARB(GLint location, GLsizei count, GLfloat *v);
```

Kde parameter *count* určuje počet elementov v poli *v*.

Podobne funkcie existujú aj pre integer premenné. Tie majú v názve namiesto f písmeno i.

GLSL pozná dátový typ matica. ten možno predávať pomocou funkcie:

```
GLint glUniformMatrix[2,3,4]fvARB(GLint location,
             GLsizei count, GLboolean transpose, GLfloat *v);
```

Hodnoty zapísané týmito funkciami sú permanentné. Ako vieme v GLSL možno uniformy iba čítať. Po opätovnom linkovaní shader programu, budú tieto hodnoty vynulované.

4 Stream processing na GPU

Stream processing je nový prístup v paralelnom spracovaní. V porovnaní so súčasnými architektúrami, stream procesory poskytujú niekoľkonásobne vyšší výkon pri rovnakej spotrebe a veľkosti čipu.

Pri stream spracovaní máme definovane 2 množiny údajov (streamy): vstupná a výstupná. Ďalej je definovaná séria operácii. Tieto operácie sú obyčajné na klasických architektúrach výpočtovo veľmi náročné. Nazývame ich **kernel** alebo kernel funkcie. Priebeh spracovania spočíva v aplikovaní operácii kernelu na každý element v streame.

V súčasnosti je optimálne písať kernel funkcie vo vyšších programovacích jazykoch podobných jazyku C. Sú to napríklad Cg (C for Graphics), GLSL (OpenGL Shading Language) alebo meta programovacie jazyky pre GPU Brook a Sh. Kompilery týchto jazykov sú uspôsobené, aby maximalizovali využitie hardvéru. Spočíva hlavne v minimalizácii prístupov čítania a zápisu do RAM a významne využívanie lokálnych cache procesora.

Stream spracovanie funguje veľmi dobre v aplikáciách spracovania obrazu, videa alebo DSP.

4.1 Paradigmy programovania

Na jednoduchom príklade ukážeme rozdiely medzi rôznymi prístupmi programovania. Máme 2 množiny dát, v ktorých po sebe nasledujúce štvorice prvkov spolu nejako súvisia. Tieto 2 množiny chceme spočítať.

4.1.1 Konvenčný, sekvenčný prístup

Klasický prístup je alokovať miesto pre vstupné údaje a výstupné pole. Následne v cykle spočítať jeden prvok za druhým.

```
for(int i = 0; i < 100 * 4; i++)
   result[i] = source0[i] + source1[i];
```

4.1.2 Paralelný SIMD prístup (SWAR)

Ako v predchádzajúcom príklade aj tu počítame sekvenčne v cykle. V tomto prípade máme navyše k dispozicii v procesore register, ktory dokáže spočítať 4 prvky súčasne (vektor).

```
for(int el = 0; el < 100; el++)
   vector_sum(result[el], source0[el], source1[el]);
```

4.1.3 Paralelný Stream prístup (SIMD/MIMD)

Idea tohto prístupu je schovať sekvenčný výpočet, ktorý prebieha v hardvéri. Ako programátor definujeme typ dát v streame a definujeme kernel funkcie. Výpočet v tomto prípade môže bežať paralelne aj na stovkách ALU jednotiek v jednom procesore.

```
streamElements 100;
streamElementFormat 4 numbers;
elementKernel "@arg0+@arg1";
result = kernel(source0, source1);
```

Tento prístup je veľmi výhodný pri počítaní jednoduchých dátových štruktúr. Vďaka jednoduchšiemu návrhu ALU ich môže byt v procesore väčšie množstvo. V kombinácii s rýchlym prístupom do cache, ponúkajú v súčasnosti až 10x vyšší výkon ako klasická architektúra. Vývoj pokračuje ďalej a aj tieto ALU začínajú byť komplexnejšie, čo v budúcnosti umožní spracovávať zložitejšie dátové štruktúry a kernel funkcie.

4.2 Príklad – Lineárna Algebra

Popíšeme spôsob implementácie jednoduchého operátora z lineárnej algebry známeho z BLAS ako saxpy(). Tento operátor počíta vektorový súčet dvoch vektorov x a y škálovaný koeficientom alfa.

$$y = y + alpha * x$$

Je to jednoduchý príklad a umožní nám ukázať koncept stream programovania na GPU. Jeho súčasťou je niekoľko krokov. Treba pripraviť grafickú pipeline, nastaviť správne OpenGL parametre. Treba zaregistrovať extensions (rozšírenia grafického akcelerátoru). Je potrebne napísať shader program (kernel funkcie). Shader program treba prepojiť s OpenGL. Všetky tieto kroky popíšeme podrobnejšie v nasledujúcich kapitolách.

4.2.1 Príprava grafickej pipeline

Výsledok nášho výpočtu budeme renderovať do framebufferu. Konkrétne použijeme Frame Buffer Object (**FBO**), ktorý nám umožní ukladať čísla s 32 bitovou presnosťou. V závislosti od verzie OpenGL bude treba zaregistrovať extensions. GL_EXT_framebuffer_object. Inicializujeme ho nasledovne:

```
GLuint fb;
void initFBO(void) {
    glGenFramebuffersEXT(1, &fb);
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
```

```
}
```

4.2.2 Príprava dát, streamy

Spracúvané dáta sa v počítači nachádzajú v niekoľkých kópiách. Sú uložené v RAM čo je CPU časť nasej aplikácie. Pred spracovaním na GPU je tieto dáta potrebne preniesť do RAM grafickej karty a to v podobe textúry. V závislosti od zvolenej funkcii OpenGL na kopírovanie textúr, v tomto procese môžu vzniknúť v RAM ďalšie kópie našich dát. Takéto značné kopírovanie významne časovo zaťažuje priebeh výpočtu. Vyber správnej funkcie závisí od konkrétnej situácie. V neskoršej kapitole si priblížime vhodný spôsob pre náš prípad.

4.2.3 Texture Target

Textúry sú dátové štruktúry primárne určené na uchovávanie obrazových hodnôt (RGB,RGBA). My budeme potrebovať typ ktorý dôkaze ukladať float hodnoty. V OpenGL môžeme použiť 2 druhy textúr. GL_TEXTURE_2D je štandardný typ v OpenGL. Rozmery tejto textúry môžu byt len mocniny 2 a indexovanie v textúre je normalizovane na interval [0,1]. Ďalším možným typom ale prístupnom iba pomocou extension alebo vo vyššej verzii OpenGL je GL_TEXTURE_RECTANGLE_ARB. Tento typ umožňuje ľubovoľné rozmery textúry a indexácia jej prvkov nie je normalizovaná.

4.2.4 Texture Format

Budeme potrebovať textúru, ktorá dokáže uchovať float hodnoty. Výrobcovia grafických kariet majú každý vlastné formáty a existuje veľké množstvo kombinácii. Využijeme ďalšiu extension **ARB_texture_float**, ktorá nám sprístupní vnútorné formáty textúr väčšiny výrobcov.

```
GLuint texID;
glGenTextures (1, &texID);
glBindTexture(texture_target,texID);
glTexParameteri(texture_target, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(texture_target, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(texture_target, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(texture_target, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
glTexImage2D(texture_target, 0, internal_format,
             texSize, texSize, 0, texture_format, GL_FLOAT, 0);
```

Predchádzajúci kód vytvorí a prípoji objekt textúry, nastaví parametre filtrovania orezania atd. a nakoniec alokuje miesto v pamäti pre dáta textúry.

Voľba správnej kombinácie CPU dát ich formátu a formátu textúry je závislá na riešenom probléme a má veľký vplyv na výsledný výkon aplikácie.

4.2.5 Namapovanie dát do textúry

Aby sme boli schopní presne kontrolovať dáta v textúre pomocou jej súradníc je treba správne nastaviť projekciu s 3D priestoru na 2D obrazovku a ďalej 1:1 mapovanie texelov, z ktorých čítame dáta, na pixely, do ktorých renderujeme. Preto treba vybrať ortogonálnu projekciu a správny viewport.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, texSize, 0.0, texSize);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glViewport(0, 0, texSize, texSize);
```

Spočítané dáta je potrebne niekam uložiť. Na ich uloženie opäť použijeme textúru. Poslúži nám FBO. Textúra môže byt v jednom momente nastavená buď iba na čítanie alebo iba na zápis. Je to dôsledok paralelnej architektúry vo vnútri GPU a jeho jednoduchej RAM logiky. Textúru do ktorej chceme zapisovať musíme pripojiť k FBO. K FBO musí byt súčasne pripojený aj offscreen framebuffer.

```
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
                          texture_target, texID, 0);
```

Prvý parameter je povinný, nasledujúce 3 identifikujú pripojenú textúru a posledný určuje level mipmap filtrovania textúry.

4.2.6 Transfer textúr medzi CPU a GPU pamäťou

Aby sme mohli presunúť dáta z CPU do GPU, treba pripojiť (bind) textúru na texture_target a zavolať OpenGL funkciu. Nemáme kontrolu nad tým ako a kedy sa dáta presunu do grafickej pamäti, je to ponechané na ovládači grafiky. Po skončení prenosu možno s dátami na CPU strane manipulovať bez ovplyvnenia obsahu textúry. Prenos zahájime nasledovným volaním.

```
glBindTexture(texture_target, texID);
glTexSubImage2D(texture_target,0,0,0,texSize,texSize,
                texture_format,GL_FLOAT,data);
```

Ukazovateľ data by mal odkazovať do pamäti s počtom 4 x texSize x texSize float hodnôt.

Štandardný spôsob v OpenGL ako načítať textúru späť do CPU časti je zavolať nasledovné funkcie. Pripojiť textúru a následne zavolať funkciu kopírovania s parametrami zvoleného zdroja, formátu textúry, odkaz do cieľovej pamäti a dátový typ hodnôt.

```
glBindTexture(texture_target, texID);
glGetTexImage(texture_target,0,texture_format,GL_FLOAT,data);
```

Ak už máme inicializovaný FBO je lepšie použiť tieto funkcie.

```
glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
glReadPixels(0,0,texSize,texSize,texture_format,GL_FLOAT,data);
```

Druhý spôsob je rýchlejší a odporúčaný výrobcami. Doba prenosov dát medzi CPU pamäťou a GPU pamäťou je významná v porovnaní s dĺžkou výpočtu na GPU, preto by počet transferov textúr mal byť minimálny.

4.2.7 Realizovanie shadera

Môžeme prehlásiť, že shader zodpovedá v stream processingu kernelu. Teda operátor násobenia vektorov bude implementovaný v shaderi. Už poznáme ako sa shader dá prepojiť s OpenGL a nasledovne riadky budú zrozumiteľné. Shader dostane cez Uniformy odkaz na textúry v grafickej karte (naše stream data), spočíta ich a uloží do defaultnej premennej gl_FragColor čo je vlastne zápis do framebuffera.

```
uniform sampler2D textureY;
uniform sampler2D textureX;
uniform float alpha;
void main(void){
  vec4 y = texture2D(textureY, gl_TexCoord[0].st);
  vec4 x = texture2D(textureX, gl_TexCoord[0].st);
  gl_FragColor = y + alpha*x;
}
programObject = glCreateProgramObjectARB();
shaderObject = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);
glAttachObjectARB (programObject, shaderObject);
glShaderSourceARB(shaderObject, 1, &program_source, NULL);
glCompileShaderARB(shaderObject);
glLinkProgramARB(programObject);
yParam = glGetUniformLocationARB(programObject, "textureY");
xParam = glGetUniformLocationARB(programObject, "textureX");
alphaParam = glGetUniformLocationARB(programObject, "alpha");
```

Záver

Výsledkom práce je teoretická príprava v oblasti negrafického využitia grafických procesorov GPU a skúška jeho schopností v počítaní základných operácií lineárnej algebry, hlavne maticové operácie. Tento hardvér ponúka vysoký výkon, ale keďže je primárne určený na zobrazovanie grafiky aj pre naše potreby je potrebné využívať grafické API. Na základe niekoľkých faktorov bolo zvolené OpenGL a programovací jazyk shaderov GLSL.

V oblasti negrafického využívania GPU v súčasnosti prebieha prudký vývoj. Jedným z prínosov tohto vývoja je stream processing. V práci sú vysvetlené jeho princípy, čo znamenajú streamy a kernel operátory. Následne je prezentovaný príklad implementácie operátora lineárnej algebry touto technikou a pomocou OpenGL API.

Použitie grafického API nie je bez komplikácii. Vyskytujú sa tu obmedzenia s nutnosťou namapovať naše potreby na grafické operácie. Každá verzia GPU procesorov má svoje špecifiká a preto ich treba brať vopred do úvahy alebo sa zamerať na jednu konkrétnu verziu. V súčasnosti ako výhodné vyzerajú procesory G80 alebo R520.

Ďalší vývoj práce bude zameraný na implementovanie zložitejších kernel operátorov. Súčasne bude prebiehať analýza možnosti použitia iného softvéru ako OpenGL API, napríklad CTM framework od ATI.

Literatura

[Ran04] Randi J. Rost, OpenGL® Shading Language, Addison Wesley, 2004, kapitoly 1.7, 2.3, 6.1-3, ISBN: 0-321-19789-5.

[Stan07] Stanford Real-Time Programmable Shading Project [online]
Dostupné z URL: http://graphics.stanford.edu/projects/shading/ (3.1.2007)

[Aga02] Agarwal, P., Krishnan, S., Mustafa, N., Venkatasubramanian, S. 2002. Streaming geometric computations using graphics hardware. Tech. rep., AT&T Labs-Research.

[Chan02] Chan, E., Ng, R., Sen, P., Proudfoot, K., Hanrahan, P., 2002. Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware. In Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware.

[Four98] Fournier, A., Fussell, D. 1998. On the power of the frame buffer. ACM Transactions on Graphics, strany 103-128.

[Peer00] Peercy, M., Olano, M., Interactive multi-pass programmable shading. In K. Akeley, Siggraph 2000, Computer Graphics Proceedings,, Annual Conference Series, strany 425–432. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[Lars01] Larsen, E., McAllister, D., “Fast matrix multiplies using graphics hardware,” *Supercomputing 2001*. Workshop on Graphics Hardware.