Debreceni Egyetem

Informatikai Kar

Komputergrafika és Képfeldolgozás Tanszék

A GPU FELHASZNÁLÁSA ÁLTALÁNOS SZÁMÍTÁSI CÉLOKRA

Témavezető:
Dr. Tornai Róbert
egyetemi adjunktus

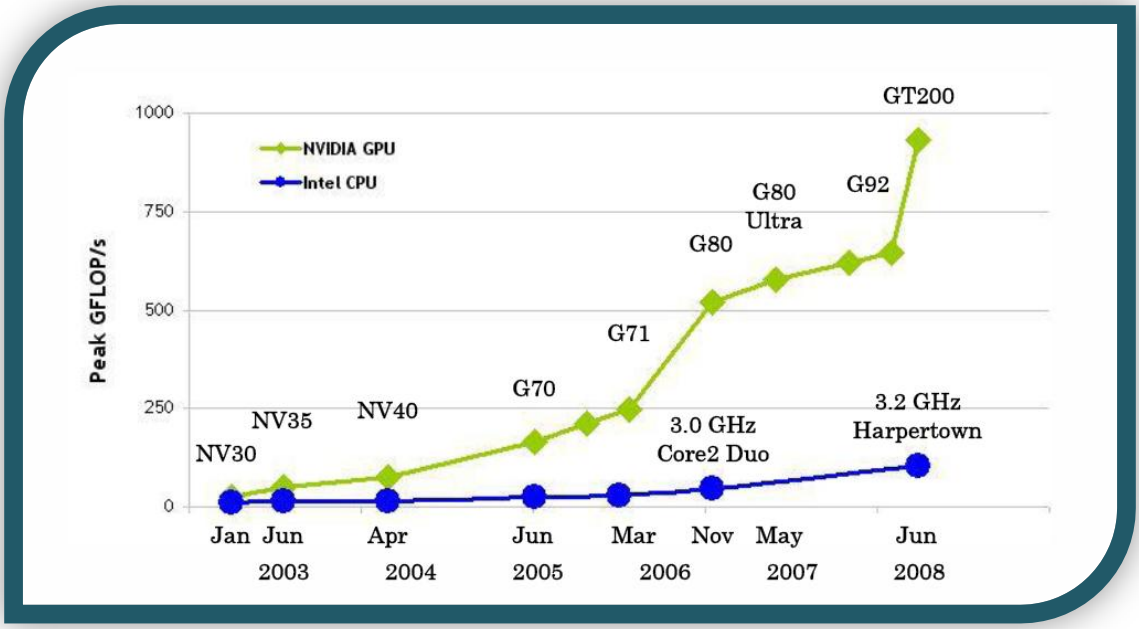
Készítette:
Lévay István
programtervező-informatikus

Debrecen
2010

Tartalomjegyzék

1. Bevezetés

Napjainkban a grafikus hardverek óriási tempóban fejlődnek. A grafikus processzorok (továbbiakban GPU) sebessége jóval felülmúlja a piacvezető általános célú processzorok (továbbiakban CPU) sebességét, és ez az előny egyre gyarapodik (1. ábra). A grafikus hardvereket ma már nem kizárólag grafikai célokra használják. Számos problémakörben alkalmazhatók, mint például sok test kölcsönhatásainak számítása, hangfeldolgozás, szélcsatorna szimulációk, de akár adatbázis-kezelésre, vagy kódfejtésre is alkalmasak. A GPU általános célú felhasználásának (angolul GPGPU – General-Purpose computation on Graphics Processing Units) zászlóshajója az NVIDIA. Ez a cég a kezdetektől fogva jelen volt a grafikus eszközök piacán; minőségi, nagy teljesítményű hardverei miatt a játékosok körében mindig nagyon népszerű volt.



1. ábra Lebegőpontos műveletek száma másodpercenként, GPU-CPU összehasonlítása [CUDA 2010]

A GPGPU térnyerésével az NVIDIA fokozatosan erre a területre összpontosított, a játékosok igényei fokozatosan háttérbe szorultak. Talán ennek, talán a technikai nehézségeknek „köszönhetően” az játékosok egyre inkább az AMD termékei felé fordulnak.

Természetesen az AMD is jelen van a GPGPU piacon saját eszközeivel, azonban az AMD mindig a grafikai igényekre kielégítésére koncentrált. Nem véletlen, hogy az utóbbi időben, mind a DirectX 10.1, mind a DirectX 11 API támogatásának tekintetében az AMD megelőzte az NVIDIA-t. Utóbbi esetben már súlyos, több mint fél éves hátrányban van az NVIDIA.

Az NVIDIA 2007 óta ad teljes körű (szoftveres, hardveres) támogatást a párhuzamosítható számítások gyorsításához. Azonban a GPGPU korszaka nem 2007-ben kezdődött, hanem több évvel korábban, amikor egyértelműen kiderült: bizonyos feladatokra a GPU sokkal alkalmasabb, mint a CPU (ez az időpont az előbbi grafikon alapján 2003-2005-re tehető). A grafikus vezérlők ekkortájt már elfogadható méretű memóriával rendelkeztek, de a GPU-k felépítése, működési mechanizmusa nem tette lehetővé az egyszerű általános célú felhasználást. (Az első programozható GPU 1985-ben jelent meg: a Texas Instruments: TMS34010.)

A dolgozat célja bemutatni a GPU-k fejlődését, és hogy az egyes fejlődési szakaszokban, milyen eszközök álltak rendelkezésre az általános célú felhasználáshoz. A téma 2003-tól kezd egyre fontosabbá válni, ezért a dolgozat ezt az időszakot fedi le (2003-2010). A dolgozatban – ha nincs más feltüntetve –, akkor alapértelmezés szerint mindig a Microsoft DirectX szoftver- és az NVIDIA hardver-termékeiről írok. A technológia színvonaláról szóló kijelentéseimet a feltüntetett időpontban, a piacon elérhető, elterjedt, polgári felhasználású eszközök alapján teszem.

A dolgozat elején bemutatom a grafikus vezérlők, és a GPU-k sajátosságait, működésüket, majd időrendi sorrendben haladva megvizsgálom, hogy programozhatóság szempontjából milyen újdonságok jelentek meg a hardver fejlődésével. Helyenként kitérek a kifejezetten hardverhez kötődő újításokra is. A dolgozat végén saját fejlesztéseket mutatok be, majd lezárásként a napjainkban elérhető, legmodernebb GPGPU eszköztárat mutatom be. Terjedelmi megszorítások miatt a programokat nem elemzem teljes részletességgel, igyekszem csak a lényeget kiemelni.

„A GPU erős számolási csodaeszköz. A GPU aritmetikai teljesítménye, egy magas szinten specializált architektúra, több éves fejlődésének eredménye. Az egyre növekvő sebesség és rugalmasság miatt sok fejlesztő leleményességének eredményeként számos olyan alkalmazás létezik, amely a GPU-t nem grafikai feladatokra használja. De sok olyan alkalmazás is létezik, amely sosem lesz képes a GPU lehetőségeit kihasználni. A szövegszerkesztés, például egy tipikusan olyan „klikkelős" alkalmazás, amely sosem fogja tudni kihasználni a GPU párhuzamosítási lehetőségeit." [John D. Owens 2005]

2. A GPU fejlődésének áttekintése

A korai számítógépekben (~1985) nem létezett GPU. Minden feladatot a központi egység végzett el, sőt a Commodore számítógépek központi egysége már videojelet küldött ki kivezetésein. A processzorok akkoriban még alacsony integráltságúak voltak, a memória pedig nagyon szűkös. Ezek a tények megmutatkoztak az akkori szoftverek grafikai képességeiben is.

A moduláris felépítésű számítógépek megjelenésével a grafikus megjelenítéssel kapcsolatos feladatok egy külön egységre hárultak. Ez a modul a grafikus vezérlő, mely kezdetben jelentéktelen rasztergrafikai támogatást nyújtott. Fontos tényező, hogy ebben a moduláris felépítésben célszerű volt a grafikus vezérlőt saját memóriával ellátni. Kezdetben a vezérlő feladata mindössze a csatolt memória feltöltése volt bizonyos tartalommal. A memória tartalma közvetlenül a megjelenítőre került, ezért a megjeleníthető kép méretét és minőségét a vezérlőkártyára telepített memória mérete korlátozta. A dedikált memória megléte kulcsfontosságú tényező, tekintve, hogy napjainkban az egyik leggyorsabb létező (tokozáson kívüli) kommunikáció a GPU és a számára fenntartott memória között zajlik. A legmodernebb, 148 GByte/sec sebességű memóriainterfész másodpercenként 30 DVD adattartalmát mozgatja meg.

A következő fontos lépés a futószalag (pipeline[1]) alapú feldolgozás bevezetése volt. Számos témakörben használatos technika ez, melynek lényege, hogy egy problémát részfeladatokra osztunk, és minden feladatra egy-egy dedikált eszközt használunk. Az eszközök egy időben dolgoznak a saját feladatukon, ezért ez a módszer igen gyors feldolgozást tesz lehetővé. Ráadásul a grafikával kapcsolatos számítások adatintenzívek, így az előbb említett dedikált eszközök sora egymás mellé redundánsan lemásolható, ezzel a futószalag „kiszélesíthető”, átbocsátó képessége növelhető.

Fontos megkülönböztetni a logikai, és a fizikai – GPU-architektúrában implementált – futószalagot. A logikai futószalag egy elméleti rendszer, amelyet 2006-ig a GPU architektúrák fizikai megvalósításukban is követtek. A legújabb GPU-k azonban már sokkal általánosabb architektúrával készülnek, így a futószalag már inkább csak logikailag, szoftveres absztrakcióként létezik.

[1] Bizonyos fordításokban csővezetéknek is nevezik.

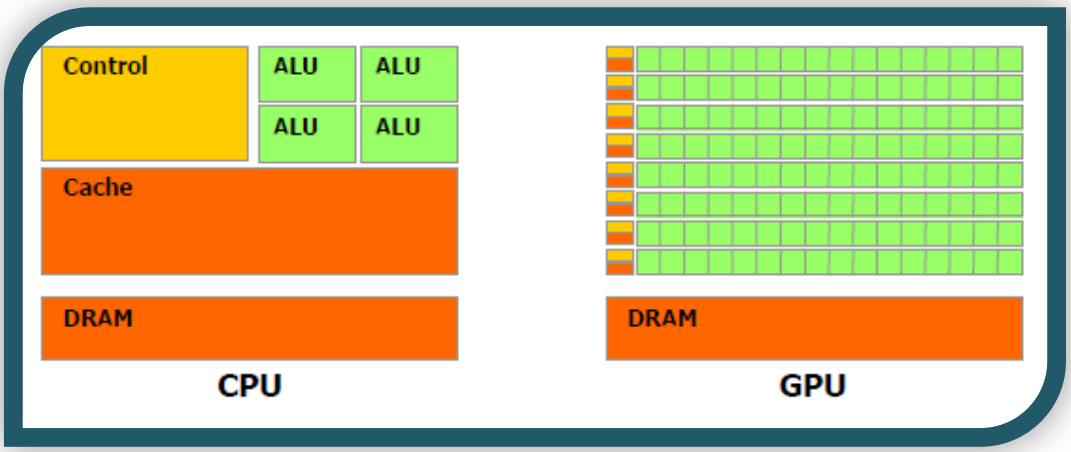
3. A GPU működése

A GPU-k nyers számolási teljesítménye napjainkban már teraFLOPS nagyságrendben, míg a CPU-k teljesítménye még mindig 10 gigaFLOPS nagyságrendben mérhető. Az GPU és a hozzá kapcsolódó dedikált memória közti kommunikáció is 5-6-szor gyorsabb, mint a CPU és a rendszermemória közötti. (A memóriák mennyiségükben is összemérhetőek.) Felmerül tehát a kérdés, mitől ilyen gyors a grafikus vezérlő, miért nem váltjuk le a központi egységet GPU-ra?

3.1. A CPU és a GPU felépítésének összehasonlítása

A kétféle processzor közti lényeges különbségek a 2. ábrán figyelhetők meg: a CPU erőforrásai nagy részét a programok vezérlésére a gyorsabb utasítás-kiválasztásra fordítja. Ilyen célra a GPU teljesen alkalmatlan. Aritmetikai logikai egységekkel (ALU) viszont a GPU van jobban felszerelve, így érthető, hogy akár két nagyságrenddel gyorsabban számol. Ennek azonban az a feltétele, hogy az egyes feldolgozó egységeken azonos utasítások fussanak.

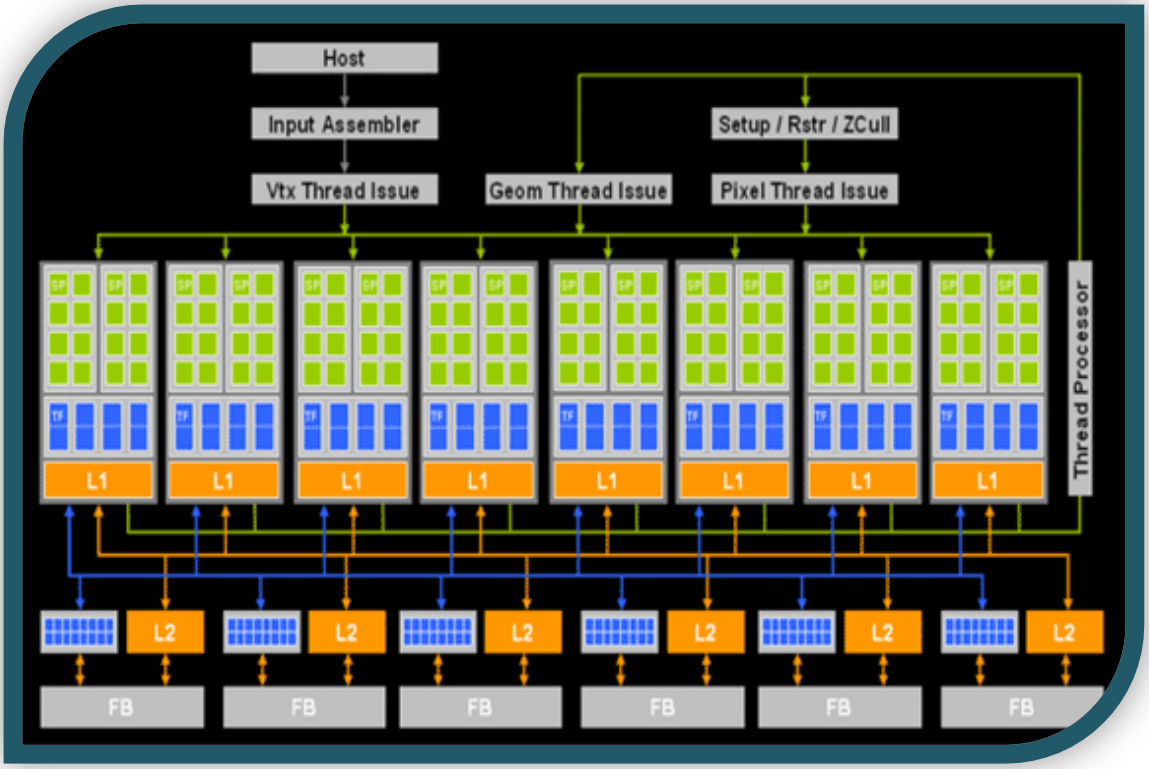
Egy program GPU-ra való portolása előtt mindig azt kell elsőként megvizsgálni, hogy az adott feladat mennyire adat- és számolás-intenzív. Ha a vezérlésátadó utasítások száma elenyésző, és nagy tömegű adaton kell elvégezni ugyanazt a számítást, akkor a portolás valószínűleg eredményes lesz.



2. ábra A CPU inkább a vezérlésre, míg a GPU a számolásra van kihegyezve [CUDA 2007]

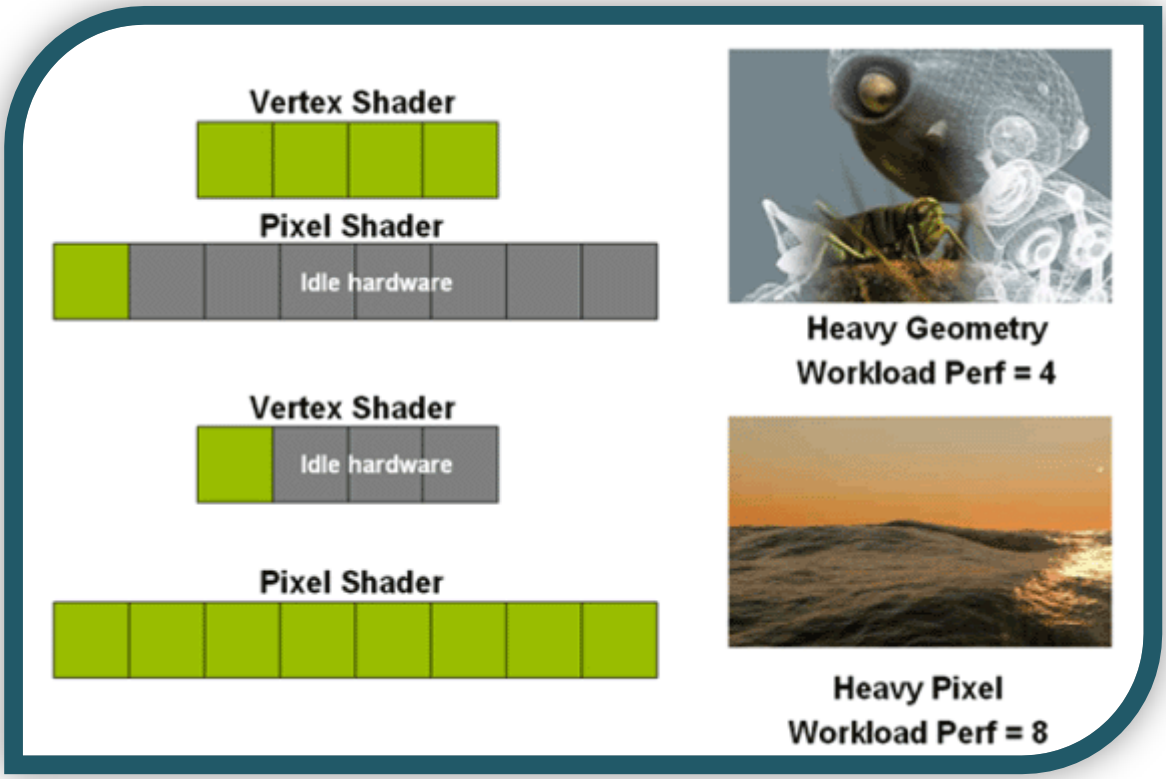
3.2. A GPU felépítése

Ahhoz, hogy jobban megértsük, milyen számításokra alkalmasak a legmodernebb GPU-k, meg kell ismernünk azok belső felépítését. A 3. ábrán a G80 processzor felépítése látható [Geforce 8800GTX 2007]. Választásom azért erre a GPU-ra esett, mert az NVIDIA jelenlegi processzorai is nagyon hasonló felépítésűek, és ez volt az első GPU, mely kifejezetten alkalmas általános célú számításokra. Az egyes összetevőkről még lesz szó a későbbiekben, itt azt érdemes megfigyelni, hogy az egyes stream-processzorok (SP) azonos méretű egységekbe tömörülnek. Minden ilyen egységhez tartozik egy kisebb számosságú textúra mintavételező (TF), valamint egy lokális gyorsítótár (L1).



3. ábra A G80-as GPU működésének vázlata [Geforce 8800GTX 2007]

„Nincsenek különálló vertex-, és pixel-shaderek, csak univerzálisak – ez az úgynevezett unified shader architektúra. Egy 3D jelenet felépítésekor egyenetlen a vertex és pixel műveletek aránya, ráadásul viszonyuk időben is változhat. Ha például fixen 4 vertex egységünk és 8 pixel egységünk lenne, akkor egy komolyabb geometriát tartalmazó grafika feldolgozásakor a négy vertex egység intenzíven dolgozna, míg a pixelesek többsége üresen járna. Fordított esetben, erősen pixelszámolásra támaszkodó helyzetben a vertex egységek lennének feladat nélkül. Az unified shader felépítés mindkét esetet elkerülhetővé teszi.” [Geforce 8800GTX 2007]



4. ábra Különböző típusú shader-processzorok kihasználtsága és kihasználatlansága különböző jelenetek renderelésekor [Geforce 8800GTX 2007]

3.3. A GPU elérése

Egy alkalmazás (Windows környezetben) két módon rajzolhat:

- Windows GDI: ez egy általános (2D) rajzolásokhoz alkalmazható API, minden Windows verzióban létezik, a grafikus ablakozás lételeme. Ez az API a grafikus vezérlő meghajtó-programjához csatlakozik.
- DirectX, OpenGL API: egységes programozási felületet biztosítanak a grafikus vezérlő eléréséhez. Fő célja a 3D-s valós idejű animációk megjelenítése.

A DirectX több rétegű felépítése lehetővé teszi, hogy grafikus vezérlőtől független, hordozható alkalmazást készítsünk. Az API egy hardver-absztrakciós réteghez (HAL) csatlakozik, ez pedig közvetlenül a vezérlő meghajtó-programjához kapcsolódik. A hardver-absztrakciós réteg jelentősége, hogy lehetővé teszi a programok tetszőleges – az API szolgáltatásait támogató –, hardveren való futtatását. A program írójának nem kell foglalkoznia a vezérlő sajátosságaival, azaz két jelentősen eltérő, de azonos szolgáltatásokat nyújtó vezérlő képes lesz módosítás nélkül ugyanazt a programot, (elvileg) azonos eredménnyel futtatni.

A grafika megjelenítése történhet HAL rétegen keresztül, ekkor a raszterizálás folyamata a hardveren zajlik le; vagy történhet referencia (REF) eszközön, ilyenkor a DirectX szoftveresen végzi el a raszterizálást, és az azt követő pixel-műveleteket. A referencia-eszköz hasznos lehet hibakeresésnél, illetve olyan funkciók kipróbálásánál, amelyet a grafikus vezérlő (hardveresen) nem támogat. Beállítástól függően a vertexek feldolgozása is történhet szoftveresen vagy hardveresen [Device Types 2007].

3.4. Fix funkciós futószalag

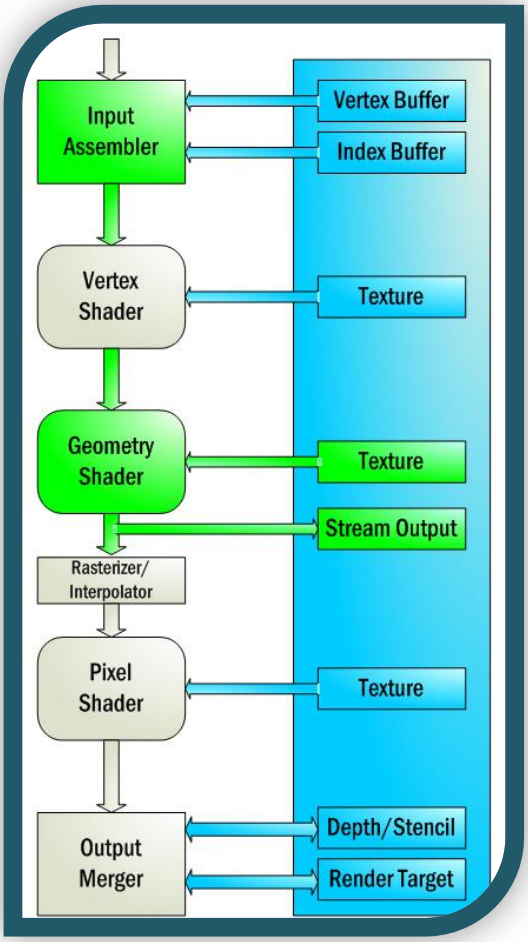
A fix funkciós futószalag nagyon jól használható grafikai célokra, jól konfigurálható. Számos olyan szolgáltatást nyújt, mely könnyebbé teszi egyszerűbb grafikus alkalmazások fejlesztését.

A fix funkciós futószalag programja az anyagok és a különböző típusú fények alapján számolja ki egy pixel színét. A játékok élethűbb megjelenítése azonban rugalmasabb futószalagot igényelt, elsőként a megvilágítás specializálásának igénye jelent meg. Ehhez egyenként kell valamilyen speciális műveletet végezni a megjelenítendő pixelen.

A fix funkciós futószalag programja a csúcspontok transzformációját végezte el beállítható transzformációs mátrixok alapján. Később a speciális effektek illetve a még gyorsabb, GPU-val támogatott feldolgozás miatt megjelent az igény csúcspontokat manipuláló programokra.

Ezen technika lényegi hátránya tehát, hogy a futószalag-rendszer csak rögzített műveletek elvégzésére volt képes. Napjainkban ez a feldolgozási mód annyira népszerűtlenné vált, hogy a DirectX 11 már nem is támogatja a fix funkciós futószalagot, annak megvalósítását a programozóra bízza. Ebből kifolyólag egy egyszerűbb háromdimenziós grafikai program megírása is igényli a flexibilis futószalag felprogramozását.

3.5. Flexibilis, programozható futószalag



5. ábra A DirectX 10 flexibilis futószalagrendszere és kapcsolódó adattárolói [Geforce 8800 GTX review 2006]

A fejlődés következő lépcsője az volt, hogy felismerték, a rugalmasság biztosításához a programozhatóságot kell megvalósítani. Természetesen a programozhatóság mértéke kezdetben minimális volt, a lehetőségek tárháza idővel egyre bővült, a korlátozó tényező a hardver volt. A programozható futószalag-rendszernek is több, egymáshoz kapcsolódó tagja van. Minden tag egy adott jellegű feladat elvégzését teszi lehetővé, de programozható módon. A fejezet hátra levő részében a Geforce 6 sorozatú grafikus vezérlő és a DirectX alapján mutatom be a flexibilis futószalag működését, illetve az egyes shadereket. A csúcspont-, és képpontfeldolgozás leírása számos idézetet tartalmaz a GPU Gems 2 című könyvből [The Geforce 6 Series GPU 2005].

A bal oldali ábrán a G80-as GPU által is támogatott DirectX10 futószalag látható. A zölddel jelölt részek támogatása hiányzik a Geforce 6 vezérlőkből, azonban fontosnak találtam említést tenni a geometriai-shaderről is.

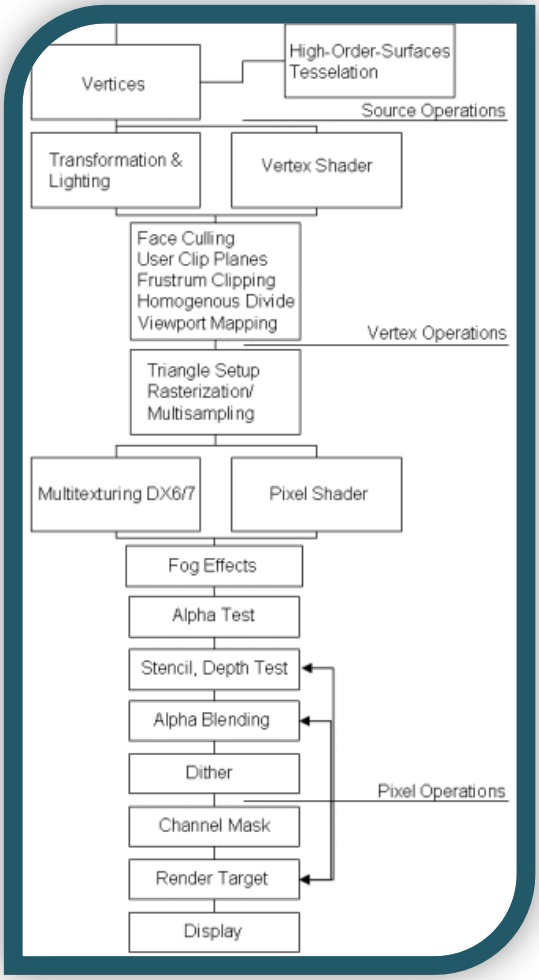
3.5.1. Csúcspont feldolgozás

Adatforrás szinten a csúcspontok összegyűjtésre kerülnek. A magas szintű primitívek moduljának feladata a magas szinten definiált primitívek N-Patch-ek, Bezier-görbék, B-spline-ok darabolása a GPU számára könnyebben emészthető háromszögekre és csúcspontokra.

A következő szinten a csúcspont operációk zajlanak. Két különböző módja van a csúcspontok feldolgozásnak. A következő két szakasz közül mindig csak az egyik lehet aktív:

1. A fix-funkciós futószalag transzformációs és megvilágítási szakasza (T&L) a renderállapotok, mátrixok, fények és anyagok paraméterezésével vezérelhető.

2. A flexibilis futószalag vertex shader szakasza programozható, így sokkal több lehetőséget nyújt, mint a T&L szakasz; azonban a megvilágítás és csúcspont-transzformációk megvalósítása is a fejlesztőre hárul.



6. ábra DirectX 8.1 futószalag-rendszer [DirectX 8.1 Pipeline 2007]

Vertex[2] processzor – Vertex-shader

A futószalag-rendszer azon programozható egysége, amely csúcspontok árnyaltabb feldolgozását teszi lehetővé. Adatforrása a vertex-buffer. A modern GPU-kon már textúra információkat is használhatunk az árnyaltabb feldolgozáshoz. Érdekes teljesítménynövelési megoldás, hogy a bemeneti adatok és a kimeneti adatok (vertexek) is gyorsítótárazva vannak, így ha egy csúcspont többször fordul elő a bemenetben (triangle-fan[3]), akkor nem fut le újra és újra a vertex-program ugyanazon csúcspontra, hanem a már kiszámított, gyorsítótárazott értéket adja vissza. A vertex-shader program csak az aktuális vertexet módosíthatja, nem hozhat létre vagy törölhet csúcspontot. Kimenetén pontosan ugyanannyi vertex jelenik meg, mint amennyi a bemenetére került. (A DirectX 8-as verziójában jelent meg a vertex shader.)

A 4. ábrán látható, hogy a hátlap selejtezés (backface culling), egyedi vágósíkok (user clip planes), nézeti gúla vágás (frustum clipping), homogén osztás (homogenous divide) és a (W2V) Window-to-Viewport transzformáció a vertex-shader után kerülnek alkalmazásra. Ezek a szintek rögzítettek, és nem programozhatóak vertex-shaderrel.

A vertex-shader bekapcsolásával automatikusan kikapcsolódnak a fix funkciós futószalag szolgáltatásai. Miért is akarnánk kikapcsolni ezeket a nagyszerűen működő, kipróbált szolgáltatásokat?

A vertex-shader rugalmasságát kihasználva a fejlesztőknek lehetőségük van többek között a következők kivitelezésére:

- procedurális geometria (ruha szimuláció, szappanbuborék effekt),
- textúra generálás,
- valós idejű perspektivikus nézet-manipuláció (lencse, víz alatti hatás),
- fejlettebb megvilágítási modellek (gyakorta a pixel-shaderrel együttműködve),
- egyéb egyedi effektek, amelyeket eddig talán senki sem implementált.

A vertex-shaderek nagyon hasznos felhasználása lehet a tömörített formájú adatok kibontása, behelyettesítése. Ezzel jelentős munkát vonhat el a CPU-tól.

[2] Csúcspont, a grafikus primitívek leírásához használt információ.
[3] (Háromszöpropeller, esernyő elrendezés) Egy közös csúcspont köré épített háromszögek sorozata.

Minden csúcspont transzformáltan, színezve hagyja el a vertex-shadert. Ekkor az úgynevezett hátlap-selejtezés (backface culling) következik: minden olyan háromszög elvetésre kerül, amely hátlappal áll a kamera felé.

Egyedi vágósíkok (user clip planes) definiálhatók, ezen síkokon kívül eső háromszögek vágásra kerülnek.

Nézeti gúla vágásra (frustum clipping) kerülnek azok a primitívek, amelyek részben vagy teljesen kívül esnek azon. A nézeti gúla úgy képzelhető el, mint egy olyan piramis, amelynek a tetején a kamera található, és az alaplap felé tekint. Ezt a gúlát csonkítja a közeli- és a távoli-vágósík, így áll elő a teljes nézeti gúla.

Ezután következik a homogén osztás (homogenous divide). Ez azt jelenti, hogy a 4 homogén koordinátával rendelkező csúcspont x, y, z komponensét elosztjuk a w komponenssel. (A homogén koordinátákra bizonyos szélsőséges vetítési esetek megoldása miatt van szükség.)

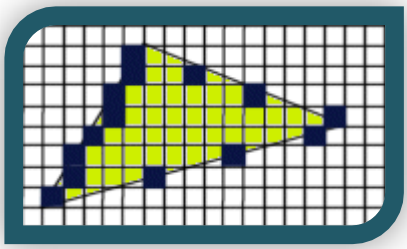
Ezek után a koordináták már úgynevezett normalizált eszköz-koordináták (normalized device coordinates (NDC)). Ezek a koordináták még átesnek egy Window-to-Viewport transzformáción, hogy a megjelenítő „világában" is észlelhetőek legyenek. Bár a megjelenítők tipikusan kétdimenziós eszközök, nem tűnik el a többi komponens (z, w), mivel később még szükség lesz rájuk.

Geometry-shader

Kizárólag a legújabb vezérlők támogatják (Geforce 8800-tól). Egy vagy több megjelenítendő objektum geometriájának „árnyalásra" használatos. Feladata, hogy futásidőben hozzon létre új vertexeket (poligonokat). Gyakorlati jelentősége, hogy ez a fázis képes újat létrehozni, azaz immáron nem a CPU adja át az objektumokat, hanem a megadott program alapján a GPU hozza létre azokat. Teszi ezt vezérlőn belül, ami jelentősen gyorsabb, mint a CPU általi létrehozás, és grafikus vezérlőre való feltöltés. Alkalmazása akkor lehet hasznos, ha egy programban a megjelenítés finomságát hardveresen kívánjuk növelni. Például egy gömb esetében a szögletességet lehet csökkenteni, új poligonok felvételével, anélkül, hogy a CPU a teljes vertex-buffert újratöltené sokkal nagyobb adatmennyiséggel. (Általános célú számításokra ezt a shadert nem szokás alkalmazni.)

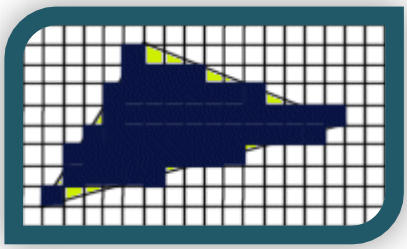
3.5.2. Képpont feldolgozás

A vertex mint transzformált, színezett csúcspont kerül a pixel-feldolgozó bemenetére. A háromszög beállítás (triangle setup) az a pont, ahol a csúcspontok élete véget ér és a pixelek élete elkezdődik. A GPU itt számolja ki a kontúrvonalak koordinátáit. Pontosabban definiálja a háromszög minden scan-vonalának első és utolsó pixelét.



7. ábra Kontúrozott háromszög, scan-vonalak beállítása
[DirectX 8.1 Pipeline 2007]

Ezután a raszterizáló interpolálja a szín- és mélységértékeket minden pixelre a csúcspontok adatai alapján.



8. ábra Kitöltött háromszög
[DirectX 8.1 Pipeline 2007]

Fragment processzor – Pixel-shader

A futószalag-rendszer azon programozható egysége, amely csúcspont-transzformációk, raszterizálás és selejtezés után előálló, textúrázáshoz előkészített képtöredékek feldolgozására szolgál. Általánosan elterjedt, hogy több fragment-processzor foglal helyet a GPU-ban.

A pixel-shadert ma már minden grafikus vezérlő támogatja. Tipikusan a geometriailag előkészített potenciális képpontok színének manipulálását végzi. A feldolgozás során 4 dimenziós vektorokkal dolgozik, melynek elemei 32 bites lebegőpontos számok. Adatforrása még a textúra-memória, melynek egy része – a nagyobb teljesítmény érdekében –, dinamikusan a GPU-ba, tokon belüli memóriába másolódik. A pixel-shader után az eredmények még átmennek néhány eljáráson, itt dől el, hogy az adott potenciális pixel bekerül-e a renderelési célba.

A következő szint a Direct3D futószalagon a ködösítés. Ennek mértékét a pixel kamerától vett távolsága befolyásolja, ezért sokkal pontosabb, mint az ún. vertex-fog ködösítés, amely a vertexeket veszi alapul a távolság meghatározásához. A ködösítés effekt vertex- és pixel-shader programban is implementálható.

Az alpha-teszt az átlátszóságot állítja be. Minden pixel alpha csatornájától függően hoz egy döntést, aminek eredményeként bizonyos pixeleket elvet. Ezek a pixelek a későbbiekben nem vesznek részt a feldolgozásban. Ez a funkció előnyös lehet általános célú felhasználásban (adatbázisban való szűrés).

A stencil-teszt maszkolja a pixelt a stencil buffer tartalma alapján. Ezt felhő és árnyék megjelenítéséhez használják játékokban.

A mélységi-teszt (depth test) Z-bufferes algoritmussal meghatározza a látható pixeleket, a takart pixelek nem kerülnek Z-bufferbe. Be lehet állítani más mélységi teszt algoritmust is, de a Z-bufferes a legelterjedtebb.

Az alpha keverés (alpha blending) a renderelési cél pixeleire keveri rá az új pixeleket.

A renderelési cél (render target) általában a back-buffer, amely a kész jelenetet viszi a kijelzőre, de a renderelési cél lehet egy textúra felülete is. Ez nagyon hasznos funkció iteratív, általános célú felhasználásnál, mivel a számolt adatokat nem kell újra képpé transzformálni egy új számoláshoz. A renderelési célt a stencil, a mélységi teszt, valamint az alpha keverési szint olvassa.

4. Magas szintű shader nyelvek

A GPU fejlődésének folyamata miatt az alacsony szintű GPU programozási nyelvek nem terjedtek el. A korai GPU-k nem voltak képesek olyan teljesítményt felmutatni, amiért érdemes lett volna azokat általános célú számolásokhoz felhasználni. A fix funkciós egységek konfigurálásához nem volt szükség GPU-kód írására. Később, a shader nyelvek megjelenésekor a cél az egyszerű, grafikus célú felhasználás volt, ezért ezek a nyelvek már eleve magasabb szintű eszközökkel jelentek meg. Az alacsony szintű programozás nagyon nehézkessé tette volna a hardver-platformok közötti átjárhatóságot, ez piaci okokból sem volt megengedhető.

A következőkben bevezetek néhány fontos fogalmat, illetve leírom a GPU programozás folyamatát. A fejezet végén a teljesség igénye nélkül bemutatok 3 magas szintű shader nyelvet. Jelentős különbség nincs köztük: mind C-alapú és hasonló fejlettségi fokú. Az egyes nyelveknél kiemelt nyelvi komponensek minden másik nyelvben is elérhetőek, legfeljebb a szintaxis különbözik.

4.1. A GPU programozás terminológiája

A következőkben az NVIDIA CUDA programozási környezetében megismerhető terminológiát mutatom be.
A grafikus vezérlő az eszköz (device).
A CPU és a rendszermemória a befogadó környezet (host).
A GPU-n futó kód a GPU-program vagy mag (kernel).
Az adatáramlás irányát a host szemszögéből szokás megfogalmazni,
azaz a rendszermemória → eszközmemória irányú adatátvitelt feltöltésnek nevezzük.

4.2. A GPU programozás lépései

Mint az látható, a modern grafikus vezérlő már nem szokványos perifériás eszköz, sokkal inkább egy gép a gépben. A processzor, a hozzá kapcsolódó dedikált memória és a programozhatóság nyomán a grafikus vezérlő teljesíti a Neumann-elvű számítógép definícióját (a soros programvégrehajtás részben teljesül).

Az adat/program betöltése azonban nem közvetlenül perifériáról, vagy háttértárról történik, hanem egy, a CPU-n futó program tölti be azokat a grafikus vezérlőbe. A GPU programozás jelenleg statikus programokkal történik, azaz memóriafoglalásra nem képes a GPU-kód. Csak előre meghatározott memóriaterületeket képes a GPU-program manipulálni. Ebből következik, hogy a foglalásokat a host-programnak kell megtennie. Tehát a host program képes eszköz- és rendszermemóriát is foglalni.

A GPU-kód betöltése is a host-program feladata. A betöltés során (GPU) gépi kódú utasításokat helyez el az eszközmemóriában. A (GPU) gépi kód előállítása mindig egy fordítóprogram feladata. A fordítás több fázisban zajlik, mindig van legalább egy köztes nyelvre való fordítás, ezután történik a specifikus alacsony szintű GPU-kód generálása.

A GPU általános célú felhasználásához tipikusan az alábbi feladatokat kell elvégezni:

- adatok feltöltése eszközmemóriába
- GPU kód feltöltése az eszközmemóriába
- GPU kód futtatása
- várakozás a GPU kód lefutására
- eredmények letöltése az eszközmemóriából.

Itt érdemes kiemelni, hogy a GPU- program indítását a host-program kezdeményezi, ám ezután a host-program azonnal folytatódhat, nem feltétlenül kell megvárnia a GPU-kód lefutását. A legmodernebb vezérlők memóriaműveleteket is képesek aszinkron módon végezni, így a GPU az adatátvitel idején is dolgozhat.

Programfejlesztési szempontból fontos kérdés, hogy lehetséges-e a GPU-n futó program hibakeresése. A válasz igen, de egyelőre csak bizonyos kompromisszumok meghozatala árán. A GPU egyszerre szálak ezreit futtatja ezért érthető, hogy nehezebben kivitelezhető a töréspontok és változó-figyelések beállítása. A legegyszerűbb módszer az emuláció: amikor a GPU-n futó kódot CPU hajtja végre. Ekkor a CPU-n futó programokhoz használt hibakereső

eszközök használhatóak. Azonban az emulátor a GPU-kódot kevesebb szálon futtatja, ezért nem tudja jól szimulálni a GPU-n való futtatást. Ez tipikusan a konkurens memóriaelérés esetén okozhat különbségeket.

Bonyolultabb hibakeresési módszer, amikor a GPU-kód valóban a GPU-n fut, és speciális eszközök segítségével kezdeményezzük a futó program felfüggesztését. Ehhez az eszköz-illesztőprogram, a hardver, a fejlesztőkörnyezet részéről is szükség lehet speciális szolgáltatásokra. Az NVIDIA megoldásának kulcsszereplője a Parallel Nsight nevű Visual Studio kiegészítés, mely lehetővé teszi a GPU-kód teljes mértékű felügyeletét. A kiegészítés egyelőre csak zárt körben érhető el, a publikus béta változat megjelenése a napokban várható.

4.3. Cg (C for Graphics)

Az NVIDIA saját fejlesztésű nyelve, egy komplex eszközkészlet elemeként jelent meg. Nagyon hasonló a Microsoft HLSL nyelvéhez. A Cg a C programnyelven alapul. A nyelvi szinten definiált típusok tekintetében van jelentősebb eltérés. A Cg fordító DirectX vagy OpenGL shader programot állít elő.

Adattípusok

Lebegőpontos (32 bites, 16 bites)
Egész (32 bites, 16 bites)
Fixpontos (12bites)
Mintavételező (sampler)
Boolean
(Void)

Vannak beépített vektortípusok és mátrixtípusok, például float3, float4. float4x4, stb...
Van struktúra és tömbtípus, ahogyan az C-ben is létezik.

Operátorok

Számos operátort támogat, így az általános aritmetikai és logikai operátorokat is.

Függvények és vezérlés

C-szerű vezérlésátadást támogat, mint az if/else, a while vagy a for. Hasonlóan a C-hez lehet függvényeket definiálni és aktuális paraméterlistával hívni.

Standard Cg könyvtár

A Cg tartalmaz néhány GPU specifikus (beépített) függvényt. Néhányuknak létezik C-beli megfelelője, mások kifejezetten GPU-programozásra valók. Ilyenek a textúra címző funkciók (tex1D, tex2D) [Cg PL 2007].

4.4. GLSL

A rövidítés az OpenGL Shading Language-ből ered. Szintén C alapú magas szintű shader-programozási nyelv. Az OpenGL ARB fejlesztette, eszköztára megegyezik a Cg-nél megismert eszközökkel. Az OpenGL API-t használja, így csak OpenGL mellet használható. Természetesen tovább viszi az OpenGL platformfüggetlenség tulajdonságát is [GLSL PL 2007].

4.5. HLSL

Ez is egy C-alapú nyelv, ami a Microsoft saját fejlesztése. A DirectX API-val együtt használható. Az alfejezet további részében Szőke munkája alapján írok [Szőke 2005].

Kulcsszavak

A kulcsszavak előre definiált azonosítók, amelyeket a HLSL nyelv fenntart, így ezeket nem használhatjuk a programunkban azonosítóként. Vannak olyan kulcsszavak is melyeket bár a nyelv fenntart, de egyelőre nem használhatóak.

Adatok

A HLSL az adattípusok széles választékát kínálja. A skalár és struktúra típusok a hagyományos magas szintű programozási nyelvekben megszokottak. Az újdonság a vektor és a mátrix adattípusok nyelvi szintű támogatásában rejlik. Továbbá a nyelv sajátosságainak megfelelően speciális típusok (textúra, mintavételező) is léteznek.

Struktúrák

A HLSL nyelv is biztosítja a lehetőséget a saját adatszerkezetek létrehozására. A struktúrák használata növeli az áttekinthetőséget, kényelmessé teszi az adatok kezelését. Struktúrát a struct kulcsszóval deklarálhatunk, a C nyelvhez nagyon hasonló módon.

Műveletek

Az adatokat operátorok és függvények segítségével dolgozhatjuk fel. Az operátorok a megszokott módon működnek a következőktől eltekintve. A HLSL esetén nyelvi szintre emelték a vektorok és mátrixok kezelését, azaz a megszokott operátorok (+,-,*,/) értelmezve vannak az adattípusokon, de fontos megjegyezni, hogy ezek elemenkénti végrehajtást jelentenek.

Tehát a * operátor nem a két vektor keresztszorzata. A mátrixok szorzatára is érvényes az a megállapítás, hogy ez a típusú szorzás nem a matematikában megszokott szorzás. A mátrixszorzatot és a kereszt szorzatot a mul() belső függvény segítségével érhetjük el. A mul() függvényen kívül a HLSL nyelv számos belső függvényt tartalmaz, melyek megkönnyítik a programozók dolgát. Nagyrészük matematikai függvény, ami a kényelmet szolgálja, míg a többi függvény a textúra adatok kinyerésére szolgál. [Szőke 2005]

5. Shader modellek, lehetőségeik és utasításkészletük

A továbbiakban a DirectX API-n és a HLSL-en keresztül mutatom be az egyes shader modellek képességeit [Asm Shader Reference 2007], majd a fejezet végén egy táblázatban foglalom össze azokat. Ezek a képességek a hardvertől erednek, ezért más gyártó szoftverén keresztül is elérhetőek (Cg, GLSL). Minden modell egy-egy fejlődési szintnek felel meg a GPU programozhatóságának tekintetében. A korai modellek még csak alapvető matematikai műveletek elvégzését tették lehetővé, erősen korlátozott mennyiségben. A modern modellek már vezérlésátadó utasításokat is tartalmaznak. A geometriai shader megvalósításához példányosító/megsemmisítő utasítások is léteznek. (Itt jegyzem meg, hogy ez az egyetlen pont, ahol a shader nyelvek statikus volta megváltozni látszik.)

A következő táblázat a DirectX verziók shader támogatását mutatja: [Shaders 2007]

DirectX Version	**Pixel-shader**	**Vertex-shader**
8.0	1.0, 1.1	1.0
8.1	1.2, 1.3, 1.4	1.1
9.0	2.0	2.0
9.0a	2_A, 2_B	2.x
9.0c	3.0	3.0
10.0	4.0	4.0
10.1	4.1	4.1
11	5.0	5.0

5.1. Model1

A mai grafikus vezérlőktől a modern szabványok nem várják el a pixel-shader 1.0 (ps_1_0) és a vertex-shader 1.0 (vs_1_0) támogatását, ezért ezeket a verziókat nem mutatom be.

Vertex-shader (vs_1_1)

A vertex-shader nagyrészt négykomponensű adatokkal dolgozik, a komponensek egy csúcspont homogén koordinátái, 32 bites lebegőpontos formában.

A következő táblázatban az 1.1 verziószámú vertex-shader utasításkészletének egy része látható.

Az *utasítás-keret*[4] oszlopban szereplő szám megadja, hogy mennyi helyet foglal az adott utasítás a maximális, 128 utasítás-keretből, a magasabb szintű utasítások több utasítás-keretet használnak.

A *beállító* oszlopban lévő x azt jelenti, hogy az adott utasítás nem végez aritmetikai műveletet, beállításra szolgál. Minden shader program első utasítása a verzió beállító utasítás.

Utasításkészlet (részlet):

Név	**Leírás**	**Utasítás-keret**	**Beállító**	**Aritmetikai**
add - vs	Összead két vektort	1		x
dcl_usage input (sm1, sm2, sm3 - vs asm)	Deklarál egy bemeneti vertex regisztert	0	x	
def - vs	Definiál egy konstanst	0	x	
dp3 - vs	3 komponensű vektoriális szorzás	1		x
dst - vs	Kiszámolja a távolság-vektort	1		x
exp - vs	Teljes pontossággal: 2^x	10		x
frc - vs	Törtrész	3		x
log - vs	Teljes pontossággal: $\log_2(x)$	10		x
m4x3 - vs	4x3 mátrix-szorzás	3		x
m4x4 - vs	4x4 mátrix-szorzás	4		x
mad - vs	Szorzás és növelés	1		x
sub - vs	Kivonás	1		x
vs	Verzió	0	x	

[4] egyes fordításokban instrukciós-keret

Pixel-shader (ps_1_1, ps_1_2, ps_1_3, ps_1_4)

A pixel-shader alapvetően képpontokkal dolgozik, fő adatforrása a textúra-buffer. Számos utasítás a textúra elérésére, képpont beállítására szolgál. A pixel-shader alapvetően négykomponensű lebegőpontos adatokon manipulál az RGBA színcsatornák miatt. Az aritmetikai utasítások ennek megfelelően nagyon hasonlóak a vertex-shader utasításaihoz, ahol a négy komponens koordinátákat jelent.

Az utasítás-keretek száma az egyes verziókban:

Verzió	Aritmetikai utasítások	Textúra utasítások
1.1	8	4
1.2	8	4
1.3	8	4
1.4 (fázisonként)	8	6

Utasításkészlet (részlet):

Verzió		**Utasítás-keret**	**1.1**	**1.2**	**1.3**	**1.4**
ps	Verziószám	0	x	x	x	x
Konstans utasítás						
def - ps	Definiál egy konstanst	0	x	x	x	x
Fázis utasítás						
phase - ps	Fázis átmenet (két fázisban lefutó kódrészlet)	0				x
Aritmetikai utasítások						
cmp - ps	Forrás összehasonítása nullával	1		x	x	x
lrp - ps	Lineáris interpoláció	1	x	x	x	x
nop - ps	Üres utasítás	0	x	x	x	x
Textúra utasítások						
tex - ps	Textúra mintavétel	1	x	x	x	
texkill - ps	Összehasonlítás alapján elvet pixeleket	1	x	x	x	x
texreg2ar - ps	Mintavételezés texturából: alpha és vörös komponensek	1	x	x	x	

5.2. Model2

Ebben a modellben megjelenik a ciklus és a vezérlésátadó utasítás, mint programozási eszköz. Az egyes vezérlők más és más tulajdonságokkal rendelkeznek a futtatható utasítások számának és a hívási lánc hosszának tekintetében.

Vertex-shader (vs_2_0)

Utasítások száma: 256 utasítás-keret van, de a futó utasítások száma ennél jóval magasabb lehet a ciklusok miatt. Ez az érték vezérlő-függő, de minimum 65535.

Fontos megemlíteni, hogy ebben a modellben a feltételes vezérlésátadás nem olyan módon történik, mint az általános programokban. A shader processzorok párhuzamos működéséből adódik, hogy a programoknak azonosan kell működniük minden processzoron. Hosszuknak, futásidejüknek azonosnak kell lenniük. Ebből adódik, hogy a döntéshozatal nem lehet dinamikus, azaz fordítási időben eldől, hogy pontosan milyen kód fut. A későbbi verziók ezt a megkötést eltörlik.

Utasításkészlet (új utasítások, részlet):

Név	Leírás	Utasítás-keret	Aritmetikai	Vezérlés-átadó
abs - vs	Abszolút érték	1	x	
call - vs	Szubrutin hívás	2		x
callnz bool - vs	Feltételes szubrutin hívás	3		x
else – vs	**else - vs** blokk kezdete	1		x
endif – vs	**if bool - vs**...**else - vs** blokk lezárása	1		x
endloop – vs	**loop - vs** blokk lezárása	2		x
endrep - vs	Ismétlő blokk lezárása	2		x
if bool – vs	if bool - vs blokk kezdete (logikai feltétellel)	3		x
label - vs	Címke (vezérlésátadáshoz)	0		x
loop – vs	Ismétlés	3		x
mova - vs	Adatmozgatás lebegőpontos regiszterből címregiszterbe (a0)	1	x	
nrm - vs	4D vektor normalizálása	3	x	
pow - vs	x^y (hatványozás)	3	x	
rep - vs	Ismétlés	3		x
ret - vs	Szubrutin vagy főprogram befejezése	1		x
sgn - vs	Előjel	3	x	
sincos - vs	Szinusz és koszinusz.	8	x	

Pixel-shader (ps_2_0)

Ebben a pixel-shader verzióban nincsenek vezérlésátadó utasítások. Azonban a korábbi szigorú korlátok enyhültek. Az utasítás-keretek száma 96 (64 aritmetikai+32 textúra). A textúra mintavételezések száma maximum 16.

Megnövekedett a felhasználható regiszterek száma is. Átmeneti regiszter: 12, Konstans lebegőpontos regiszter 32, textúra-koordináta regiszter 8.

Az utasításkészlet jónéhány – a vertex-shadernél bemutatott – utasítással bővült. Pár új textúra és beállító utasítás is megjelenik ebben a verzióban.

Utasításkészlet (új utasítások, részlet):

Név	Leírás	Utasítás-keret	Beállító	Aritmetikai	Textúra
abs - ps	Abszolút érték	1		x	
dcl (sm2, sm3 - ps asm)	Deklarál egy összerendelést a vertex-shader kimeneti és a pixel-shader bemeneti regiszterei között	0	x		
m4x3 - ps	4x3 mátrix szorzás	3		x	
m4x4 - ps	4x4 mátrix szorzás	4		x	
max - ps	Maximum	1		x	
min - ps	Minimum	1		x	
nrm - ps	Normalizálás	3		x	
pow - ps	x^y (hatvány)	3		x	
rcp - ps	Reciprok	1		x	
sincos - ps	Szinusz és koszinusz	8		x	
texld ps_2_0 and up	Textúra mintavétel	1			X
texldb - ps	Textúra mintavételezés. részletessége a w komponenstől függ.	1			X

Vertex-shader (vs_2_x)

Bár változás csak a verziószám minoráns részében van, az újítás mégis jelentős. Az eszköztár a dinamikus feltételes vezérlésátadás lehetőségével bővül, azaz immáron nem csak konstans regiszterek lehetnek a feltételes vezérlésátadó utasítások paraméterei. A program maximális hossza nem változott. Továbbra is 256 utasítás-keret van. Az átmeneti regiszterek száma megnőtt.

Azonban nem minden eszköz támogatja azonos mértékben a dinamikus feltételes vezérlésátadást. A vezérlésátadás mélysége, azaz a hívási lánc hossza is erősen eszközfüggő: 0 és 24 között változhat.

A vezérlésátadásnak tehát 2 fő fajtája van: a statikus, ami a 2.0 verzióban jelenik meg; illetve a dinamikus, mely ennek a verziónak az újdonsága.

Létezik egy nagyon érdekes döntéshozó utasítás: a predikációs utasítás. Használatához külön predikációs regiszter létezik, mely egy 4 komponensű logikai regiszter. A predikációs utasítás

előtt ennek értékét be kell állítani. A regiszter tartalma alapján a kívánt utasítás csak a megfelelő komponensértékekkel fog manipulálni [Predicate 2007].

Formája:

```
([!]p0[.swizzle]) instruction dest, srcReg, ...
```

Ahol:

- [!] opcionális logikai negálás (NOT)
- p0 a predikátum-regiszter
- [.swizzle] Komponensfelcserélés (svindli)
- Tetszőleges aritmetikai- vagy textúra-utasítás. Nem lehet statikus, vagy dinamikus vezérlésátadó utasítás.
- dest, srcReg, ... az utasítás által használt regiszterek.

Tegyük fel, hogy a predikátum-regiszter tartalma rendre igaz, igaz, hamis, hamis, amit például a következő kód biztosít:

```
// adott r0 regiszter. értékei= 0,0,1,1
// adott r1 regiszter. értékei= 1,1,0,0
setp_le p0, r0, r1 //predikációs regiszter beállítása, less-or-equal feltétellel
(p0) add r2, r3, r4 //predikációs regiszter alapján összeadás
```

A fenti kód hatása:

```
r2.x = r3.x + r4.x
r2.y = r3.y + r4.y
```

Az r2 regiszter z és w komponensei nem változnak, mivel a predikációs regiszter hamis értékeket tartalmaz a z és w komponensek helyén. A predikációs utasítás egy utasítás helyet foglal az adott aritmetikai- vagy textúra-utasítás mellett.

Pixel-shader (ps_2_x)

Hátrányát ledolgozandó, megjelenik a statikus és dinamikus vezérlésátadás és a predikáció. (Ezek a vertex-shaderben megismert módon működnek, ezért itt külön nem részletezem.) Új utasítások kerülnek bevezetésre, többek között gradiens számoláshoz. Az utasítás-keretek száma 96-512 között változhat az adott hardvertől függően. (A futó utasítások száma jóval nagyobb lehet.) Nincs külön megszorítás a textúra-utasítások számára. A textúra mintavételezések száma max. 16. Az átmeneti regiszterek száma 32.

5.3. Model3

Ebben a modellben a textúra elérésre és a még korlátlanabb programozásra törekedtek az alkotók. Megjelenik a textúra elérés vertex-shaderben is. Az utasítások száma alig bővül, azonban a shader program maximális hossza már több mint elég: 512 utasítás-keret. Általánosan elmondható, hogy 40 utasításnál hosszabb shader programot nem érdemes írni, inkább több fázisban kell rövid programokat futtatni. (40 utasítás rosszabb esetben is csak 320 utasítás-keretet használ, így az 512-es felső határ nagyon kényelmes.)

Vertex-shader (vs_3_0)

Ez a verzió már hozzáférést enged a textúrákhoz, amit eddig csak a pixel-shader programok tehettek meg. A temporális regiszterek száma 32. A programok maximális hossza hardverfüggő, de minimum 512 utasítás-keret. A bemeneti, és a kimeneti regiszterek is indexelhetők ciklusváltozóval. Megjelenik a mintavételezési regiszter (a textúra elérés miatt szükséges). Az utasításkészlet nem bővült jelentősen.

Pixel-shader (ps_3_0)

Új regiszterek jelennek meg: előlap-, tisztán lebegőpontos szín- és pozíciósregiszter. A program hossza 512 utasítás-keret vagy még több lehet. Az utasításkészlet nem bővült jelentősen.

5.4. Model4

Ez a modell rengeteg újdonságot hoz. Visszafelé a Model2-ig kompatibilis. Csak a legújabb vezérlők támogatják. A programozás még korlátlanabb, az utasításkészlet még bővebb. A futószalag-rendszer egy szinttel bővül: a geometriai shader a vertex-shader és a pixel-shader között foglal helyet. Célja, hogy a jelenetet új elemekkel bővítse, vagy módosítsa azt [Model4 2007].

A Model4-et támogató (NVIDIA) vezérlők esetében fontos megemlíteni, hogy fizikailag már nem különül el a pixel-, vertex- és geometriai-shader, hanem azonos tudású processzorok dolgoznak együtt. (Nem a futószalag szakaszaként értelmezett pixel-shader vagy vertex-shader szűnt meg, hanem csak a megvalósításukkal foglalkozó dedikált processzorok.) Az egységesített hardverarchitektúra szoftveres oldalról abban is megmutatkozik, hogy az egyes shaderekben ugyanazok az utasítások, szolgáltatások érhetők el. A programok hosszára nincs megkötés. Az utasításkészlet elég bőséges tetszőlegesen bonyolult számolások elvégzéséhez.

5.5. Model5

Ez a modell a DirectX 11-ben jelenik meg, a Model4 bővítése. Több új futószalag szakasz is megjelenik ebben a verzióban:

- Hull-shader
- Domain-shader
- Compute-shader

Előbbi kettő a tesszaláció megvalósításához szükséges, tehát kimondottan grafikus célú. A harmadikról a későbbiekben lesz szó DirectCompute témakörben.

Ez a modell már nagyon modern, általános célú számolásokra kizárólag a compute-shaderrel érdemes felhasználni.

5.6. Modellek áttekintése

Áttekintésképpen táblázatosan mutatom be az egyes modellek pixel- és vertex-shadereinek lehetőségeit. (Általános célú számolásokra ezeket a shader típusokat csak a Model2, Model3 és Model4–ben célszerű felhasználni, ezért a táblázatok csak ezeket a változatokat tartalmazzák [Shaders 2007].)

Pixel-shaderek összehasonlítása:

	PS_2_0	**PS_2_a**	**PS_2_b**	**PS_3_0**	**PS_4_0**
Függő textúra korlát	4	Korlátlan	4	Korlátlan	Korlátlan
Textúra utasítás korlát	32	Korlátlan	Korlátlan	Korlátlan	Korlátlan
Pozíciós regiszter	Nincs	Nincs	Nincs	Van	Van
Utasítás-keretek	32 + 64	512	512	≥ 512	≥ 65536
Futó utasítások	32 + 64	512	512	65536	Korlátlan
Textúra indirekció	4	Korlátlan	4	Korlátlan	Korlátlan
Interpolált regiszterek	2 + 8	2 + 8	2 + 8	10	32
Utasítás predikáció	Nincs	Van	Nincs	Van	Nincs
Bemeneti indexelt regiszterek	Nincs	Nincs	Nincs	Van	Van
Átmeneti regiszterek	12	22	32	32	4096
Konstans regiszterek	32	32	32	224	16x4096
Önkényes komponenscsere	Nincs	Van	Nincs	Van	Van
Gradiens utasítások	Nincs	Van	Nincs	Van	Van
Ciklusszámláló	Nincs	Nincs	Nincs	Van	Van
Előlap regiszter (kétoldalas megvilágításhoz)	Nincs	Nincs	Nincs	Van	Van
Dinamikus vezérlésátadás	Nincs	Nincs	Nincs	24	Van
Bitműveletek	Nincs	Nincs	Nincs	Nincs	Van
Egészek (IEEE-integer32)	Nincs	Nincs	Nincs	Nincs	Van

- **PS_2_0** = DirectX 9.0 eredeti **Shader Model 2** specifikáció.
- **PS_2_a** = NVIDIA GeforceFX-re optimalizált modell.
- **PS_2_b** = ATI Radeon X700, X800, X850 –ra optimalizált modell, DirectX 9.0b.
- **PS_3_0** = **Shader Model 3**.
- **PS_4_0** = **Shader Model 4**.

Vertex-shaderek összehasonlítása:

	VS_2_0	**VS_2_a**	**VS_3_0**	**VS_4_0**
Utasítás-keretek	256	256	≥ 512	4096
Futó utasítások	65536	65536	65536	65536
Utasítás predikáció	Nincs	Van	Van	Van
Átmeneti regiszterek	12	13	32	4096
Konstans regiszterek	≥ 256	≥ 256	≥ 256	16x4096
Statikus vezérlés-átadás	Van	Van	Van	Van
Dinamikus vezérlés-átadás	Nincs	Van	Van	Van
Hívási lánc hossza	Nincs	24	24	Van
Vertex-textúra elérhetőség	Nincs	Nincs	Van	Van
Textúra mintavevők	0	0	4	128
Geometriai példányosítás	Nincs	Nincs	Nincs	Van
Bitműveletek	Nincs	Nincs	Nincs	Van
Egészek (IEEE-integer32)	Nincs	Nincs	Nincs	Van

- **VS_2_0** = DirectX 9.0 eredeti **Shader Model 2** specifikáció.
- **VS_2_a** = NVIDIA GeforceFX-re optimalizált modell.
- **VS_3_0** = **Shader Model 3**.
- **VS_4_0** = **Shader Model 4**.

Látható, hogy az alacsony verziószámú modellek között gyártófüggőek is előfordulnak, de a többi esetében is szoros megszorítások vannak érvényben. Egyedül a 4.0 verziónál látható, hogy a megszorítások csak elvileg léteznek, a gyakorlatban egyetlen program sem fogja feszegetni azokat.

6. Általános számítások HLSL alapokon

Az elméleti áttekintés, a szoftveres háttér ismertetése után bemutatok egy elterjedt megoldást, amellyel rá lehet bírni a grafikus processzort, hogy „olyan képet alkosson, aminek adattartalma egy kiindulási feladat megoldása”.

A GPU-t grafikus shader programozással csak konverziók után lehet bevonni a számolásba. Például hangfeldolgozás esetén a hangot előbb képpé kell alakítani, ezután a megfelelő shader-programot futtatni a GPU-n majd az eredményt ismét vissza kell alakítani hanggá.

Ilyen célra a legelterjedtebb shader a pixel-shader. Ha ezzel a shaderrel szeretnénk egy általános problémát megoldani, akkor a forrásadatokat „képpé” (textúra) kell alakítani. Egy textúrának azonban több rétege is lehet, így egy texelhez[5] többször 4×32bit adat tartozhat (minden texelhez tartozik 4 színcsatorna). Ha elkészült a megfelelő textúra, akkor létre kell még hoznunk egy (vagy több) poligont, amire majd a textúra felfeszül. Ezek után biztosítanunk kell azt, hogy a poligon pontjai meg is jelenjenek a pixel-shader bemenetén. Ez úgy érhetjük el, hogy a poligont előlappal pontosan a kamera elé helyezzük, ám ezzel még nem biztosított, hogy a pixel-shader feldolgozza az összes neki szánt pixelt. A pixel-shader csak raszterizálás után kapja meg az adatokat, így ha a célpoligon távol van a kamerától, annak csak néhány pixele fog a raszterre kerülni. Ezt elkerülendő olyan vetítést, illetve olyan Window-to-Viewport transzformációt kell alkalmazni, hogy a forrástextúra minden eleme pontosan egyszer jelenjen meg eredményként a renderelési célban. (Az előbbiekben feltételeztem, hogy nincs olyan eljárás (vágás, per-pixel-fog), amely a poligon(ok) megjelenését megváltoztatná.) Ezek után a poligon(ok) biztosan bekerülnek a renderelési célba, ahonnan ki lehet nyerni az eredményképet, és vissza lehet alakítani az eredeti formába, vagy akár mint textúrát vissza is lehet tölteni és újra fel lehet dolgoztatni, esetleg egy másik shader-programmal.

Vannak azonban bizonyos korlátok, amikkel számolni kell a fenti folyamatban. Például a textúrák kiterjedése korlátozott, a shader programban a textúra-elérési utasítások száma limitált lehet. A textúra elérése – ha az nincs gyorsítótárazva – lassú a processzor feldolgozási sebességéhez képest, így a processzor várakozik.

[5] A textúra egy alkotóeleme

Fontos tudni, hogy a GPU-k általában SIMD (Single Instruction, Multiple Data) utasítás-végrehajtással működnek, azaz minden processzor más-más adaton dolgozik, de azonos utasítással.

A pixel-szintű döntéshozó vezérlés-átadást az egyes GPU-k más-más módon kezelik. Az egyes shader utasítások egyszerre pixelek százain hajtódnak végre, így előfordulhat, hogy egy feltételes elágaztatás után minden pixelen más-más utasítások következnének. Kérdés, hogy ilyen esetben hogyan biztosítja a GPU, hogy minden pixelen a neki megfelelő ág fusson le és mégis minden pixelre azonos utasítások hajtódjanak végre.

Vannak vezérlők, amelyek úgy oldják meg a problémát, hogy ha a feltétel akár egyetlen pixelen is hamis, akkor az összes pixelen a hamis ág fut le. Egyértelmű, hogy ez a viselkedés elfogadhatatlan általános célú számolásoknál. (Grafikában néhány pixel enyhe torzulását okozza csak.)

A Geforce 6800 vezérlő GPU-ja ebben az esetben a feltétel mindkét ágának utasításain „végigvezeti” az összes pixelt, de minden pixelhez csak a neki megfelelő ágat érvényesíti (Csak a megfelelő ágban engedélyezi a regiszter-írást). Ez a működés napjaink GPU-ira is jellemző.

Összefoglalásképpen elmondható, hogy a shaderek lehetőségei korlátozottak, különböző hardveren különböző eredményt adhatnak. Célszerű észben tartani a képpont feldolgozás lépéseit, hogy bizonyosak lehessünk afelől, hogy a pixel-shader feldolgozza az összes neki szánt adatot.

6.1. Hello World!

Készítettem egy alkalmazást, amely pixel-shader 2.0 támogatással számolja ki 65536 szám négyszeresét. Ez a program nagyon primitív és olyan problémát old meg, amire nem célszerű GPU-t használni, azonban egy jól áttekinthető képet nyújt az általános célú felhasználás gyakorlati kivitelezéséről.

A számolás előtt feltöltök egy tömböt float típusú adatokkal. A tömb feltöltése során figyelembe kell venni, hogy ez a tömb, egy textúra adatforrása lesz, így a mérete optimálisan 256×256×4. A textúra mérete 256×256, mivel a grafikus vezérlők ezt preferálják, a négyes csoportok pedig a 4 színcsatorna miatt szükségesek.

```
float* rgbadata = (float*)malloc(256*256*4*4);
      for (int i=0; i<256*256; i++)
            {
            float ChannelA= 0.9f;
            float ChannelB= 1.0f*((float)i/(256*256));
            float ChannelG= 0.0f;
            float ChannelR= 0.0f;
            memcpy(&rgbadata[i*4+0],&ChannelR,4);
            memcpy(&rgbadata[i*4+1],&ChannelG,4);
            memcpy(&rgbadata[i*4+2],&ChannelB,4);
            memcpy(&rgbadata[i*4+3],&ChannelA,4);
            }
```

A tömb feltöltése után feltöltöm az adatokat a textúrába. Ehhez zárolni kell a teljes textúrát, majd memóriamásolást kell végrehajtani.

```
tolock.top=tolock.left=0;
tolock.bottom=tolock.right=255;
//Zároljuk a RAM-ban lévő texturát
g_pPreTexture->LockRect(0,&rect,&tolock, D3DLOCK_DISCARD);
//Nyers adatmásolás a textúrára
memcpy(rect.pBits,rgbadata,256*256*4*4);
//Textúra feloldása
g_pPreTexture->UnlockRect(0);
```

Ezután a videó-memóriában létrehozott textúrát frissítem, a RAM-ban tárolt textúra adataival.

```
g_pd3dDevice->UpdateTexture(g_pPreTexture,g_pTexture);
```

A textúra beállítása után létre kell hozni többek között egy renderelési célt, azt a helyet, ahová a renderelés eredménye kerül:

```
g_pd3dDevice->CreateRenderTarget(256,256,D3DFMT_A32B32G32R32F,
D3DMULTISAMPLE_NONE,0,true,&g_pRenderTarget,NULL);
```

A renderelési cél mérete és formátuma megegyezik a létrehozott textúrákéval.

A renderelés folyamán beállítom a shader programot, a texturát és a renderelési célt:

```
g_pd3dDevice->SetPixelShader(g_pPixelShader);
g_pd3dDevice->SetTexture(0, g_pTexture);
g_pd3dDevice->SetRenderTarget(0, g_pRenderTarget);
```

Az utolsó sor elhagyásával a képernyőn, grafikaként jelenik meg az eredmény, ekkor azonban nem nyerhető ki a számszerű érték.

A renderelés után az eredmények pontosan úgy nyerhetők ki a renderelési célból, mint ahogy korábban a textúrába írás történt; csak fordított irányú adatmásolással.

A pixel-shader kódja a következő:

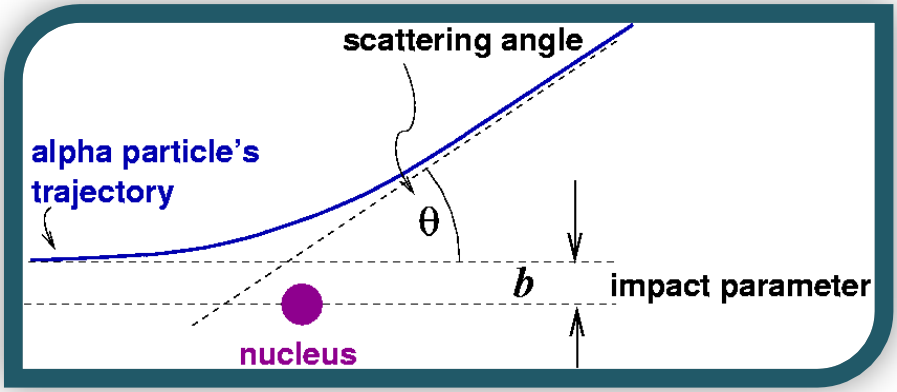
```
PS_OUTPUT ps_main( in PS_INPUT In )
{
	PS_OUTPUT Out;
	Out.Color=4* tex2D(Tex0, In.Texture);
	//Egy textúra-mintavételezés után visszaadja
	//a texel „színének négyszeresét”.

	return Out;
}
```

A program forráskódja a csatolt adathordozón megtalálható.

6.2. Rutherford-szórás

A következő példaprogram az előző továbbfejlesztése. Ez már tartalmaz néhány haladó fogást is. A feladat: egy részecske (alpha particle) mozgását kell szimulálni, annak szóródását kell mérni. A részecske egy másik, nagyon nagy tömegű részecskével (nucleus) van kölcsönhatásban. A kis tömegű részecske adott magasságban (b), adott vízszintes irányú sebességgel halad, majd a Coulomb erő hatására a nagy tömegű részecske közelében pályát módosít. A pálya hajlásszöge (θ) jellemzi a részecske indításának magasságát (9. ábra).



9. ábra A Rutherford szóródási kísérlet vázlata [Rutherford 2003]

A feladat gyakorlati megvalósítása során egyszerre 1024 részecske mozgását fogom szimulálni. Minden részecske más-más magasságban indul, ebből kifolyólag másképp fognak szóródni. (Kölcsönhatás mindig csak egy kistömegű és a nagytömegű részecske között jön létre.)

A szimuláció folyamán a kísérlet teljes idejét szeletekre osztom (nem azonos hosszú időszeletekre!). Minden időszelet elején ismert a részecskék sebessége és pozíciója. A pozícióból számolható a rá ható erő, abból pedig a gyorsulás. A gyorsulás és az időszelet méretét ismerve meghatározható a részecske pozíciója és sebessége az időszelet végén (egyenletesen gyorsuló mozgást feltételezve).

Minden egyes renderelés egy-egy időszeletet fog számolni. A renderelés végeredménye, a következő renderelés forrása is egyben. A megoldás során úgy járok el, hogy ez az adat-visszaforgatás az eszközmemóriában történjen, ezzel jelentős időt spórolva. (Az eredmény helyességének ellenőrzése miatt minden rendereléskor letöltődik adat az eszközmemóriából.)

Ebben az esetben két adatforrásra lesz szükség. Az egyik textúra a pozíciós adatokat tartalmazza, míg a másik a sebesség adatokat. A textúrák létrehozása, feltöltése az előző programban megismert módon történik.

A megvalósítás során dupla bufferelést alkalmaztam, azaz 2x2 textúra létezik az eszközmemóriában, a renderelés elején 2 textúra adatforrásként, 2 pedig renderelési célként viselkedik. A renderelés után megcserélem a 2 pár textúrát.

```
if (Swap)
      {
      g_pd3dDevice->SetTexture( 0, g_pTexturePos ));
      g_pd3dDevice->SetTexture( 1, g_pTextureVelo ));
      }
      else
      {
      g_pd3dDevice->SetTexture( 0, g_pTexturePos2 ));
      g_pd3dDevice->SetTexture( 1, g_pTextureVelo2 ));
      }
```

A Swap logikai változó értéke minden rendereléskor átbillen. Fontos, hogy a renderelési cél referenciákat mindig meg kell semmisíteni és ki kell nullázni. Enélkül a renderelési cél változatlan lesz az első textúra hozzárendelés után.

```
if (g_pRenderTargetPos!=NULL )
      {
      g_pRenderTargetPos->Release();
      g_pRenderTargetPos=NULL;
      };
if (g_pRenderTargetVelo!=NULL )
      {
      g_pRenderTargetVelo->Release();
      g_pRenderTargetVelo=NULL;
      }
```

A renderelési cél mindig a másik két textúra lesz:

```
if (Swap)
      {
      g_pTexturePos2->GetSurfaceLevel(0,&g_pRenderTargetPos));
      g_pTextureVelo2->GetSurfaceLevel(0,&g_pRenderTargetVelo));
      }
      else
      {
      g_pTexturePos->GetSurfaceLevel(0,&g_pRenderTargetPos));
      g_pTextureVelo->GetSurfaceLevel(0,&g_pRenderTargetVelo));
      }
```

A shader-kód a mozgásegyenletet implementálja:

A textúra-mintavételezéshez már nem elegendő az alapértelmezett működés. Explicit módon kell meghatározni a mintavételezőket.

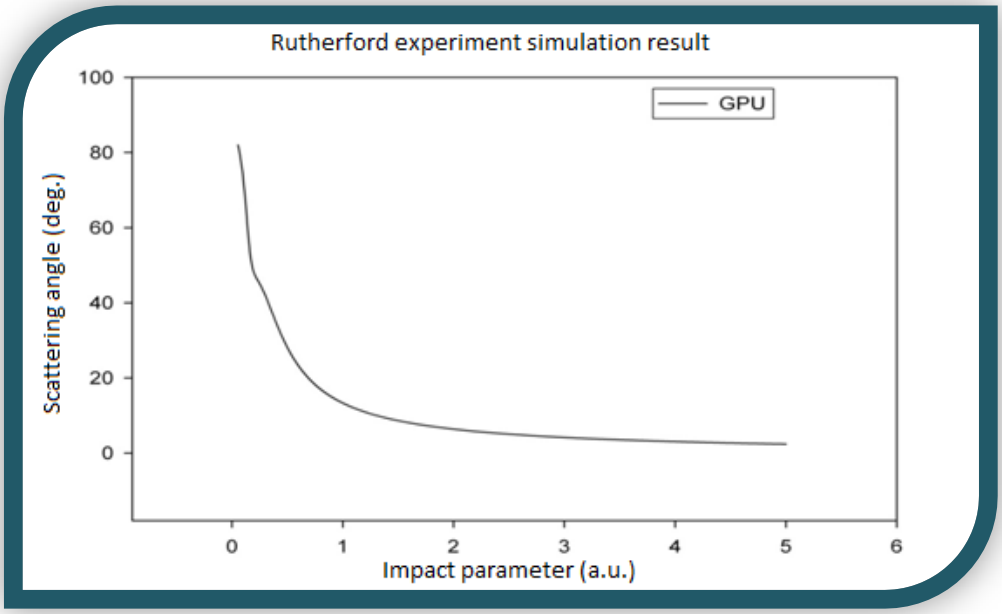
```
texture PosTexture : TEXTURE0;
sampler TexPos = sampler_state
      {
      Texture = (PosTexture);
      };

texture VeloTexture : TEXTURE1;
sampler TexVelo = sampler_state
      {
      Texture = (VeloTexture);
      };
```

A shader-program kódja textúra mintavételezéssel kezdődik, a mozgásegyenlet egyenletesen gyorsuló, egyenes vonalú mozgást ír le. A kódból az is kiderül, hogy nem egységes időszeletekre van osztva a szimuláció idő, hanem az egyes részecskék időszeletenként megtett útja van meghatározva (0,5 atomi egység). Erre azért volt szükség, hogy a kísérlet eredménye egyszerűbben kiértékelhető legyen. Jelen szimulációban a nagy tömegű részecske tömege végtelen nagy, és az origóban helyezkedik el, töltése: 1 atomi egység. A kis tömegű részecske tömege 1 atomi egység, töltése 1 atomi egység. Az erőhatások a Coulomb-törvény szerint kerülnek kiértékelésre.

```
PS_OUTPUT ps_main( in PS_INPUT In )
{
      PS_OUTPUT Out;
      float4 pos=tex2D(TexPos, In.Texture);
      float4 velo=tex2D(TexVelo, In.Texture);
      float dist= length(pos);
      float4 force = (-1.0f /float(dist*dist))*normalize(pos);
      float4 acc= force / 1.0f;
      float accl=length(acc);
      float velol=length(velo);
      float step=0.5f;
      float time= (-velol+sqrt(2.0f*accl*step+
      float(velol*velol)))/accl;
      Out.Color0=pos+velo*time+acc*0.5*float(time*time);
      Out.Color1=velo-acc*float(time);
      return Out;
}
```

Az 10. ábra a Rutherford-szórási kísérlet GPU-n számolt eredményeit mutatja. Az eredmény az elvárásoknak megfelelő.



10. ábra a Rutherford szórási kísérlet szimulációjának eredménye

A program forráskódja a csatolt adathordozón megtalálható.

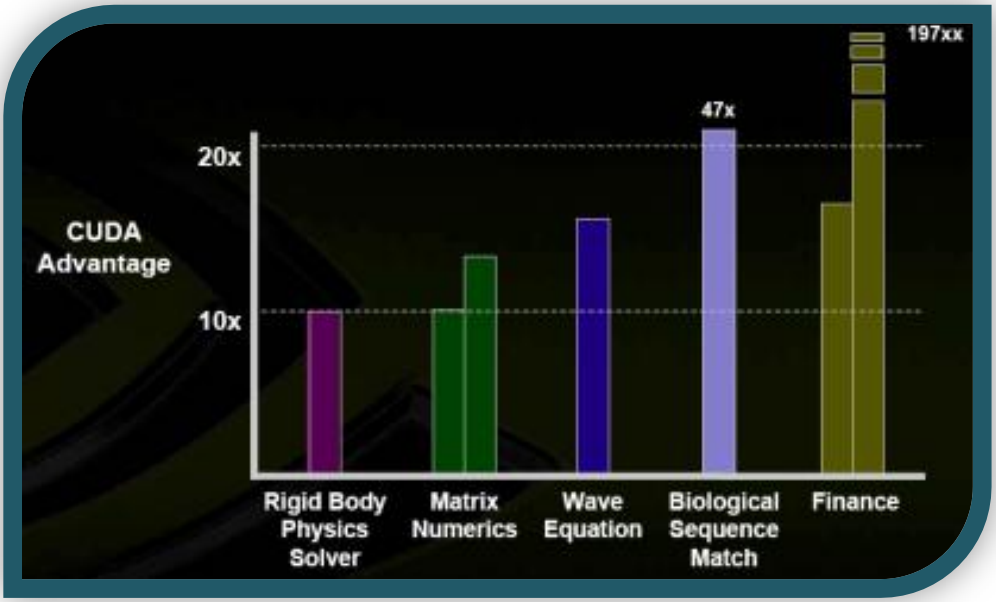
7. Magas szintű, általános célú nyelvek

Ebben a fejezetben bemutatom azokat a nyelveket és eszközöket, amelyek lehetővé teszik, hogy az általános célú számítások elvégzéséhez ne kelljen a GPU-val „elhitetni”, hogy képpel dolgozik. Ennek a megközelítésnek az előnye, hogy nem kell transzformációt végezni a bemeneti/kimeneti adatokon, így a program sokkal rövidebb lehet; illetve a programozás jóval szabadabb, mint a grafikus shaderek esetében.

7.1. CUDA

A CUDA (Compute Unified Device Architecture) az NVIDIA eszközrendszere általános problémák GPU-n való megoldásához.

A technológia fiatalságát jól mutatja, hogy az 1.0-ás verzió hivatalos kézikönyve 2007. június 23-án jelent meg. A következő néhány oldalon ez alapján mutatom be az architektúrát [CUDA 2007].



11. ábra A CUDA-val elért sebességnövekedés különböző számolási feladatokban [CUDA Performance 2008]

A CUDA széles körben alkalmazható eszköz. Munkáim során eddig sokrészecske-szimuláció [Lévay 2008/1], illetve orvosi képfeldolgozás [Lévay, 2008/2] felgyorsítására használtam. Előbbi esetben két nagyságrenddel sikerült gyorsítani a számolást, hasonló eredménnyel jártam 3D képfilterek esetében is: csaknem ezerszeres sebességnövekedést sikerült elérni.

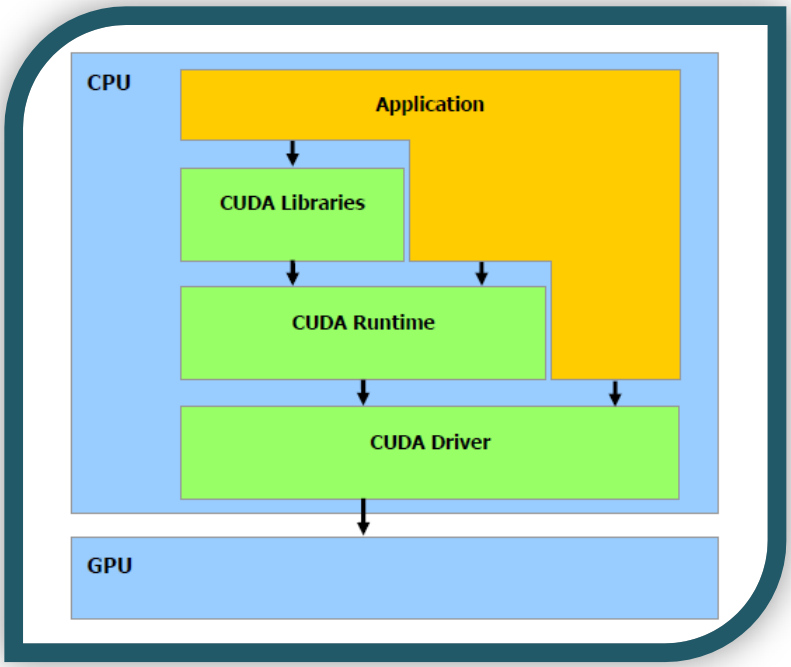
Azonban az is bebizonyosodott, hogy a GPU alkalmazása nem minden esetben növeli meg azonos mértékben a feldolgozás sebességét: 3D képregisztráció során például „csak" 50-szeres, PET-rekonstrukció esetén pedig 5-10-szeres sebességnövekedést sikerült elérni.

A CUDA jelenleg a következő eszközökkel használható:

- Geforce 8 széria,
- Quadro FX 5600/4600,
- Tesla solutions,
- Geforce 9 széria,
- Geforce 100,
- Geforce 200 ,
- Geforce 300 ,
- Geforce 400 széria.
- Várhatóan az összes ezután megjelenő NVIDIA hardver.

Emulátor segítségével kevésbé felszerelt gépen is kipróbálható, igaz ekkor a sebességnövekedés elmarad; sőt akár nagyságrendekkel is lassabb lehet a GPU program emulátoros futtatása, mint ugyanazon program CPU-s implementációjának futtatása. Az emulátort hibakereséshez és a működés „elvi" helyességének igazolására célszerű használni. Mivel az emuláció CPU-n fut, ezért egyrészt a rendszerben lévő GPU típusa lényegtelen, másrészt natív hibakeresési funkciók is elérhetőek.

Egy CUDA alapú alkalmazás több rétegen keresztül éri el a GPU-t:
hardvereszköz-meghajtó, driver-API, futtatókörnyezet-API és két magasabb szintű általános célú függvénykönyvtár (CUFFT és CUBLAS). A hardvert már eleve úgy tervezték, hogy könnyűsúlyú meghajtó programmal és futtatókörnyezettel működjön együtt, ezzel is megnövelve a szoftveres oldal átbocsátóképességét.



12. ábra Az architektúra szoftveres rétegződése [CUDA 2007]

A CUDA lehetővé teszi az eszközmemória tetszőleges címzését, így nagyobb programozási szabadságot nyújt az adat-gyűjtési és az adat-szórási műveleteknél. Programozási szempontból ez annyit jelent, hogy az eszközmemória tetszőleges része írható-olvasható, csakúgy, mint a CPU-n. A GPU-kód illetéktelen eszközmemória-hozzáférését nem felügyeli operációs rendszer, ezért az drasztikus következményekkel járhat: eszköz-illesztőprogram leállása, a monitoron megjelenő kép torzulása, teljes rendszerleállás. Ezeket a problémákat a CUDA fejlesztői – vélhetőleg – fokozatosan megpróbálják majd kiküszöbölni.

Az GPU on-chip osztott memóriával is fel van szerelve, amely nagyon gyorsan írható és olvasható. Ezt a szálak egymással való kommunikációra használják. Ezzel az eszközmemóriához fordulások száma csökkenthető, márpedig az eszközmemória a GPU-feldolgozás szűk keresztmetszete.

7.1.1. Programozási modell

A programozási modellben – a már megismertek mellett – az alábbi fogalmak használatosak:

Multiprocesszor (MP): egy utasításszámlálón osztozó stream-processzorok összessége.

Shared memory: célja az egy blokkban (lásd: később) lévő szálak közti gyors adatcsere megvalósítása.

Kernel: GPU-n futó kód (mag).

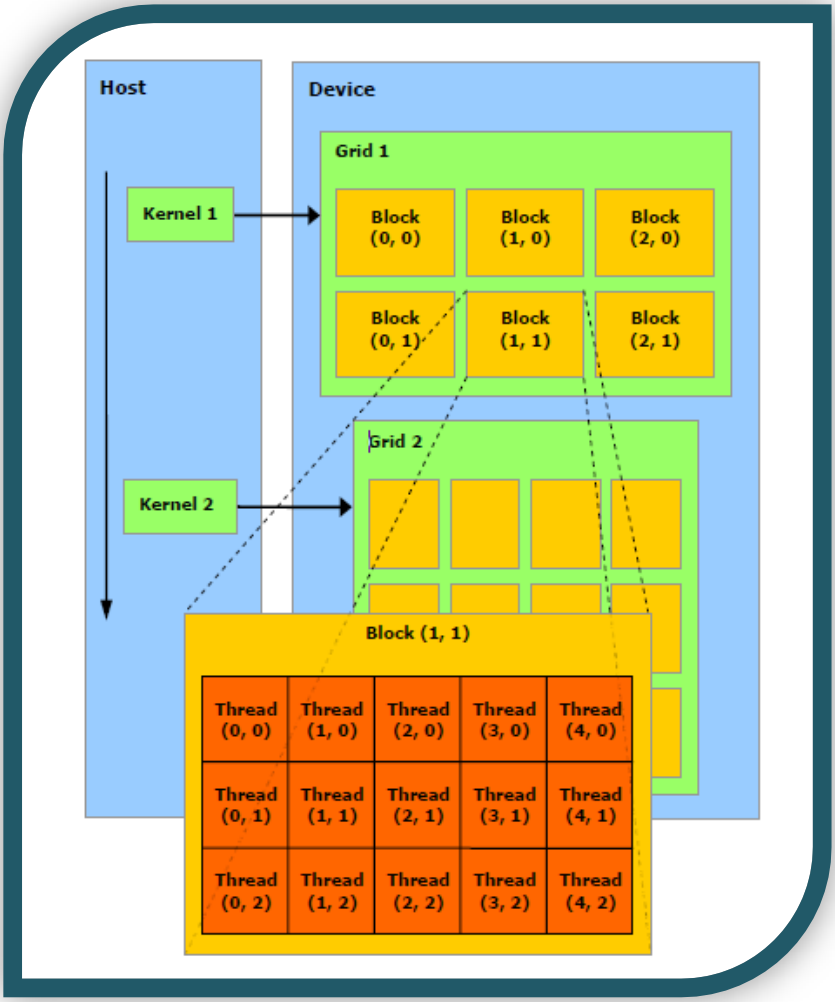
Globális memória: az eszközmemória megnevezése GPU-program kontextusban.

Szálak kötegelése

A GPU-n futó szálak több szinten kötegelődnek. Minden szál egy blokknak a része. Több blokk egy rasztert alkot. A GPU-n több aktív raszter futhat.

A blokk a szálak egy kisebb egysége. Az egy blokkban lévő szálak közös gyors elérésű memóriaterületen osztoznak (shared memory). Szinkronizáltan képesek együttműködni. Minden szál azonosítható a threadID-vel, ami a blokkon belül egyedi. A könnyebb címezhetőség kedvéért a szálak több dimenziós tömbbe rendeződve alkothatnak egy blokkot. A dimenzió szám 1, 2 vagy 3.

Van egy korlát, aminél több szálat egy blokk nem tartalmazhat (jelenleg 512). De az azonos méretű, azonos kerneleket futtató szál-blokkok kötegelhetők, így az adott magot futtató szálak száma jóval magasabb lehet. Ennek azonban az az ára, hogy a magasabb szinten kötegelt szálak kevésbé tudnak együttműködni, mivel az egy raszterben, de külön blokkban futó szálak nem tudnak osztott memórián kommunikálni illetve szinkronizálni. Ez a modell lehetővé teszi, hogy a mag újrafordítás nélkül képes legyen futni több feldolgozóval felszerelt eszközön. Az eszköz dönthet úgy, hogy az egyes blokkokat egymás után futtatja, vagy – ha van elegendő erőforrása – több szál-blokkot is indíthat (automatikus skálázás). Minden blokk a *blockID*-val azonosítható, a szálakhoz hasonlóan 2 dimenziós tömbbe is rendezhetőek, így létrehozva egy rasztert (13. ábra).

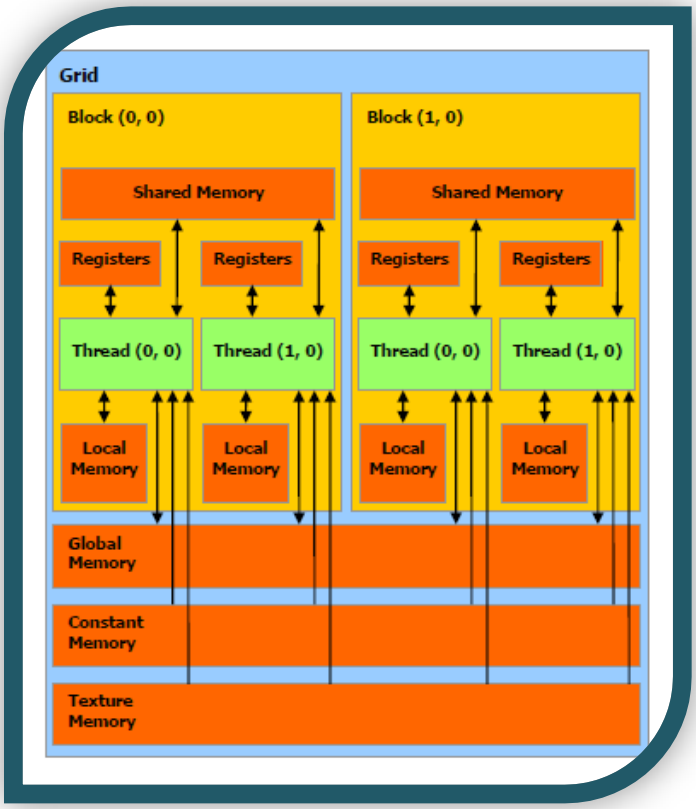


13. ábra A CUDA szálkötegelése [CUDA 2007]

7.1.2. Memória modell

Egy, az eszközön futó kernelnek kizárólag az eszközmemóriához és az on-chip memóriához van hozzáférése, az alábbi módokon:
(A mennyiségi adatok a GT200 GPU-ra vonatkoznak.)

- szálanként saját regiszter (erősen korlátozott mennyiségben 8192byte/MP),
- szálanként saját helyi-memória (programozástechnikai megoldás a regiszterek kiváltására, valójában globális memória),
- blokkonként osztott-memória (16Kbyte/MP),
- raszterenként globális memória,
- raszterenként konstans memória (csak olvasható, másolat készül on-chip-re 64Kbyte/MP),
- raszterenként textúra memória (csak olvasható, másolat készül on-chip-re 64Kbyte/MP).



14. ábra A CUDA memória-szervezése [CUDA 2007]

7.1.3. A nyelv

A CUDA kifejlesztésekor cél volt, hogy lehetőleg olyan nyelvet alkossanak, amely jelentéktelen tanulást igényel C programozásban jártas fejlesztők számára.

Az elérhető, illetve készíthető függvények 3 csoportba sorolhatóak:

- Kiszolgáló oldal (host): olyan függvények, amelyek a számoló eszközök elérését, lekérdezését, beállítását végzik.
- Eszköz oldal (device): olyan függvények, amelyek az eszközön futnak.
- Közös (global): általános könyvtár, közös típusokkal és szolgáltatásokkal. A C szabvány-könyvtár azon elemei, amelyek az eszközön és a kiszolgálón is képesek futni.

A nyelv hozzáadott típusminősítőket tartalmaz, amelyekkel az adott programozási eszköz felhasználási területe adható meg. (__device__ , __host__ , __global__)

A futtatás konfigurálása

A kernel futtatásához speciális szintaxis tartozik. A konfiguráció definiálja a raszter- és a blokk-dimenzióinak számát, és azok méretét. A szintaxis: <<<Dg, Db, Ns>>> formájú kifejezés beillesztve a (globális) függvény neve és az aktuális paraméterlista közé.

Ahol a Dg a raszter méretét definiáló 3 elemű vektor. A méretet a Dg.x*Dg.y értékek határozzák meg. Ennyi blokk lesz egy raszterben.

Db hasonló módon az egy blokkon belüli szálak számát adja meg.

Az Ns az osztott memória kívánt méretét adja meg, és dinamikusan lefoglalja azt. Nem kötelező megadni.

Például a

```
__global__ void Func(float* parameter);
```

függvény hívása:

```
Func<<< Dg, Db, Ns >>>(parameter);
```

A nyelv és a technológia további részleteinek kifejtése jóval túlmutat jelen dokumentum keretein.

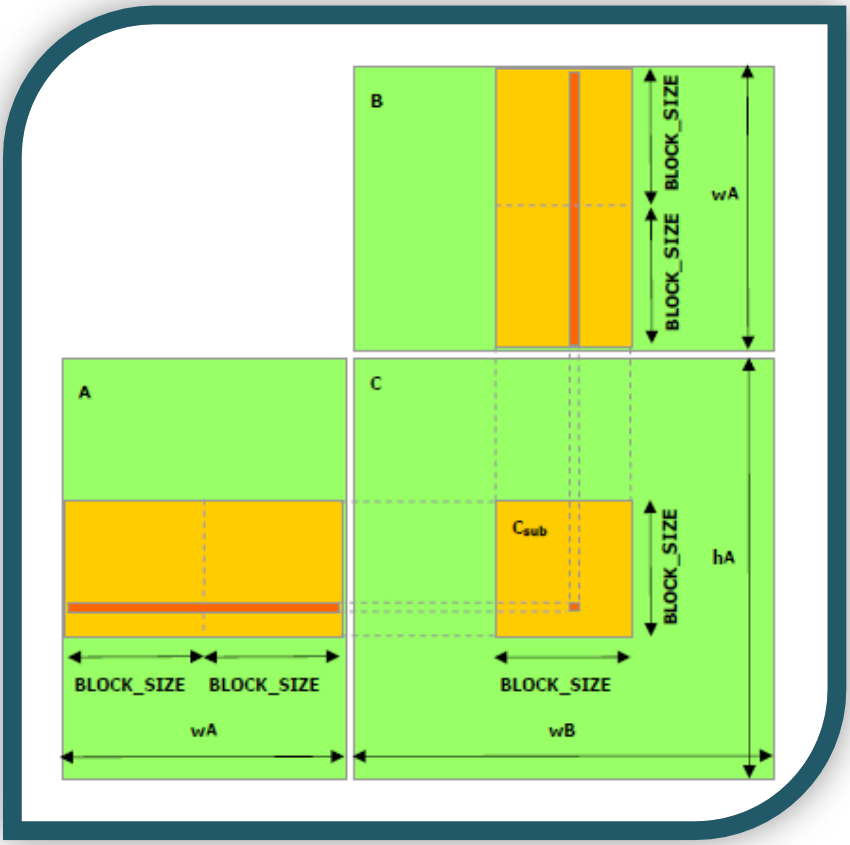
7.1.4. CUDA esettanulmány

Ebben a részben egy példaprogram alapján bemutatom, hogyan kell a GPU lehetőségeit optimálisan kiaknázni. A példaprogramban is látható, itt már nincs szükség arra, hogy elhitessük a GPU-val, hogy képpel dolgozik, tetszőleges adatszerkezet használható adatforrásként.

A program célja: két adott méretű mátrix szorzása.

A program forrása: [CUDA 2007].

A következő ábra mutatja a feladat vizuális megközelítését:



15. ábra Mátrix-szorzási feladat vázlata [CUDA 2007]

A mátrixszorzás több szálra bontható a következőképpen:

- Minden egyes szál-blokk egy C_{sub} négyzetes rész-mátrix kiszámítását végzi.
- Minden egyes szál a C_{sub} mátrix egy elemének kiszámítását végzi.

A programban használt konstansok számszerű értékei:

- BLOCK_SIZE: 16
- wA („A” mátrix szélessége): 2×BLOCK_SIZE
- hA („A” mátrix magassága): 3×BLOCK_SIZE
- wB („B” mátrix szélessége): 3×BLOCK_SIZE

Így egy C_{sub} számolásán egy blokkban dolgozó szálak száma 256 (16×16).

Ahogy az ábrán látható, a kiindulási mátrixban az aktuális C_{sub} mátrix alapján olyan mátrix jelölődik ki, amelynek mérete a C_{sub} mátrix méretének egész számú többszöröse. Az ábrán látható „A” mátrixban így 2 db BLOCK_SIZE×BLOCK_SIZE méretű mátrix keletkezik, míg a „B” mátrixban szintén kettő. Az algoritmus egy iterációban mindig kijelöli az „A” illetve „B” mátrix megfelelő négyzetes rész-mátrixát. Ezekkel, mint bemeneti adatokkal elvégzi a szorzást, az eredményekkel egy átmeneti tárolóban lévő mátrix elemeit növeli. (Az ábrán látható esetben az iteráció kétszer fog lefutni.)

Miért van erre szükség? Miért nem lehet egy lépésben meghatározni a C_{sub} mátrix elemeit?

Lehetne, de akkor minden esetben a globális memóriához kellene fordulni, ami jelentős időveszteség, tekintve, hogy a feladat jelentős része memóriaolvasásból áll. (A globális memória elérése két nagyságrenddel több időt vesz igénybe, mint az osztott memória elérése. Itt emelném ki, hogy ez a „lassú” memóriaelérés is 100 GByte/sec körüli értéket jelent.)

A NVIDIA által szolgáltatott megoldás jól átgondolt. Az egyes C_{sub} töredékek meghatározása több lépcsőben zajlik, így a memóriaszükséglet (jelen esetben negyedére) csökken. Ez a kis mennyiségű adat elfér a blokk osztott memória-tartományában, így nem kell minden egyes elem számolásához az eszközmemóriához fordulni.

Mivel egy blokk csak egy C_{sub} töredéket számol, ezért nyilvánvalóan több blokkot kell futtatni párhuzamosan (jelen esetben kilencet).

A mag forráskódja

```
int bx = blockIdx.x;
int by = blockIdx.y;
int tx = threadIdx.x;
int ty = threadIdx.y;
```

Az „A”, „B” mátrixok C_{sub}-hoz tartozó „sávjainak” címzése:

```
int aBegin = wA * BLOCK_SIZE * by;
int aEnd   = aBegin + wA - 1;
int aStep  = BLOCK_SIZE;
int bBegin = BLOCK_SIZE * bx;
int bStep  = BLOCK_SIZE * wB;
```

Átmeneti tároló az adott szálban kiszámolt C_{sub} értékhez:

```
float Csub = 0;
```

A kijelölt sávokban a BLOCK_SIZE×BLOCK_SIZE méretű részmátrixok megkeresése, sávok bejárása:

```
for (int a = aBegin, b = bBegin;
      a <= aEnd;
      a += aStep, b += bStep)
      {
```

Az „A” illetve „B” mátrixban kijelölt kvadratikus mátrixok betöltése osztott memóriába:

```
__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
AS(ty, tx) = A[a + wA * ty + tx];
BS(ty, tx) = B[b + wB * ty + tx];
```

Szálak szinkronizálása, ezen a ponton minden szál betöltötte a saját adatát:

```
__syncthreads();
```

Sor-oszlop kompozitum képzése, az eredmény növeli a C_{sub} változót. Nem értékadás, mivel a következő fázis(ok) még hozzáadódnak.

```
for (int k = 0; k < BLOCK_SIZE; ++k)
      Csub += AS(ty, k) * BS(k, tx);
```

Szinkronizáció. Minden szál elvégezte a számolást.

```
__syncthreads();
} // Következő „A”, „B” részmátrix feldolgozása.
```

Az ebben a szálban kiszámolt eredmény beírása a globális memóriába:

```
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
```

7.2. OpenCL

A Khronos Group által kifejlesztett nyílt, platform- és eszközfüggetlen szabvány heterogén rendszerekhez (tipikusan erősen párhuzamosított rendszerekhez).

Az eszközfüggetlenség előnye, hogy ha rendelkezésre áll megfelelő GPU, akkor azon is futtatható a kód. Ha nem, akkor pedig ugyanaz a kód – változtatás nélkül – futtatható CPU-n is. A szabvány nem zárkózik el más eszközök használatától sem.

Első változata 2009 második felében jelent meg. Szálkezelési, memóriakezelési modellje azonos a CUDA-nál ismertetettel; a programozási felület gyakorlatilag csak a függvények nevében tér el. A nyelvi elemek is igen hasonlók.

Első ránézésre nagyon tetszetős a szabvány, gyakorlati alkalmazása során azonban számos problémába ütközhetünk:

- Az eszközfüggetlenség csak a központi szolgáltatásokra korlátozódik. A szabvány megengedi bővítmények használatát. Azonban nem minden eszközön érhető el minden bővítmény, azaz speciális programok esetén már nem áll fenn az eszközfüggetlenség.
- Tipikusan teljesen másképpen kell megírni egy párhuzamos eszközre optimalizált, és egy soros feldolgozásra optimalizált programot. Ezért ha a GPU nem áll rendelkezésre, akkor egy arra optimalizált program valószínűleg jóval lassabban fog futni CPU-n, mint ugyanazon program soros feldolgozásra optimalizált változata.
- Az OpenCL implementációkat az eszközök gyártói szállítják, jellemzően az eszköz-illesztőprogramokkal. Az implementáció Windows rendszeren egy DLL fájlban tárolódik (opencl.dll). Ha egy számítógépben két gyártó eszközei is megtalálhatóak, akkor is csak egy DLL lesz elérhető a rendszerben, ez a DLL pedig csak az egyik eszközt tudja majd „meghajtani”.
- Megfigyelhető, hogy az eszközökben rejlő lehetőségek kiaknázására szolgáló bővítmények csak később kerülnek megvalósításra. Azaz a hardverben már megvan a tudás, de nem tudjuk használni, mert a megfelelő szoftver még nem készült el hozzá. (Ez a helyzet a jövőben vélhetően javulni fog.)
- Az NVIDIA mindig preferálni fogja a saját eszközét (CUDA), ezért az egyes extra funkciók jóval hamarabb fognak elérhetővé válni CUDA-ban, mint OpenCL-ben.

7.3. DirectCompute

A DirectX 11 egyik újdonsága. Nem teljesen eszközfüggetlen, csak a GPU általános célú felhasználására lett tervezve. Egyedül Windows platformon érhető el, csak olyan grafikus vezérlőkkel amelyek DirectX 11 kompatibilisek.

Előnye, hogy nagyon szorosan integrálódik a Direct3D renderelési egységéhez, így nagyon könnyen összeköthető a grafika és a hozzá tartozó általánosabb célú számolás. Például egy játékban a robbanás élethű megjelenítéséhez az objektum vertexei könnyen megfeleltethetőek pontszerű testeknek. A testek szóródása DirectCompute-tal számolódik, de mivel ezek a testek egyben egy grafikus objektum vertexei is, így a képernyőn egy robbanás effekt lesz az eredmény. A módszer előnye, hogy nem kell adatmásolást végezni, pusztán más értelemben fogjuk meg a csúcspontokat. Természetesen a CPU-t nem terheli ez a számolás.

A működési elv ez esetben is nagyon hasonló a CUDA-hoz, a szálkezelési-, és memória-modell szinte azonos. DirectCompute-ban a GPU programozás nyelve a HLSL, tehát a már jól ismert nyelvet használja. Hiányosságként megemlíthető, hogy a pointereket nem támogatja. Ezzel ugyan biztonságosabb a memória elérése, de nyilvánvalóan jelentős hátrányokat is okozhat. Hasznos szolgáltatás, hogy a tömbök méretét figyeli, nem engedi meg a tömb túlcímzését, ezért viszont valószínűleg teljesítményveszteséggel kell számolnunk.

Általánosságban elmondható, hogy célja a grafika és a fizikai számítások hatékony, egyszerű összekapcsolása. (Itt jegyzem meg, hogy az OpenCL-hez is elérhetők DirectX/OpenGL együttműködést biztosító bővítmények, illetve a CUDA a kezdetektől fogva nyújt támogatást ezekhez a grafikus API-khoz.)

8. Összefoglalás

Jelen műben sikerült bemutatnom, hogy a GPU általános célú felhasználása nem olyan futurisztikus vagy félelmetes dolog, mint azt sokan gondolják. A GPU programozása tényleg teljesen más felfogást igényel, és nem árt, ha az embernek vannak tapasztalatai párhuzamos programozás terén. Azonban egyértelműen kijelenthető, hogy napjainkban jóval egyszerűbb a GPU általános célú programozása, mint pár évvel ezelőtt. Jól kell ismerni a párhuzamosítani kívánt algoritmust is, hiszen a mátrixszorzásos példa is megmutatta, hogy az algoritmus sajátos tulajdonságait jól ki lehet használni. Szerencsére ma már a legfontosabb algoritmusok párhuzamos megfelelőjét is ismerjük, a többmagos processzorokon (CPU) a gyakorlatban is régóta alkalmazzuk őket. A GPU azonban jóval túlmutat egy 2-4 vagy akár 8 magos CPU lehetőségein: aktív szálak ezreit képes kezelni, olyan adatátviteli sebességgel, mely a CPU környezetében még jó ideig nem lesz elérhető.

A kutatásaim során egyértelműen kiderült, hogy a gyártók közül az NVIDIA már-már behozhatatlan előnyre tett szert ebben a szegmensben. A napokban megjelent Fermi architektúra újabb mérföldkő a GPU programozás útján. Ez a processzor-család már 64 bites pontossággal számol, támogatja az ECC hibakezelést, és sok más hasznos funkciót tartalmaz. A vezérlőre szerelt memória mérete jócskán meghaladja az 1 gigabájtot.

A CUDA fejlődése is figyelemre méltó, hiszen a 3.0-ás verzió már szinte a teljes C++ eszközkészletet támogatja. (Későbbi verzióban a virtuális tagfüggvények támogatása is várható.) A fejlesztőeszközök terén újdonság a napokban megjelenő Parallel Nsight Visual Studio kiegészítő. Ez az eszköz nagyban növeli majd a produktivitást.

A régimódi grafikus shader alapú általános célú felhasználás felett eljárt az idő. A felsorolt modern megoldások (CUDA, OpenCL, DirectCompute) sokkal egyszerűbbé teszik a GPU alkalmazását, számos olyan szolgáltatást nyújtanak, melyet képtelenség lenne hagyományos módon pixel- és vertex-shaderekkel kiváltani.

Összegzésképpen megállapítható, hogy a **DirectCompute** a jövőben a játékfejlesztők körében lesz népszerű, mert erősen támogatja a Direct3D-vel való együttműködést, illetve védelmi funkciókat is tartalmaz.

Az **OpenCL** a jövőben az AMD favoritja lesz, mivel a gyártó nem fog behódolni az NVIDIA-nak, egyelőre azonban az OpenCL jelentős lemaradásban van a CUDA-hoz képest. Felhasználása minden területen lehetséges, ahol a CUDA is alkalmazható.

A **CUDA** egy széles körben használható eszközrendszer. A NVIDIA pedig a játékosok igényeitől elfordulva egyre jobban a GPGPU-ra koncentrál. Nem véletlen, hogy a Fermi már számos olyan újítást tartalmaz, melyet a játékok – grafikai célokra legalábbis – nem fognak kihasználni. Kérdés, hogy a GPGPU piacon van-e akkora kereslet, mely pótolhatja az elhanyagolt játékosok hiányát. Egyelőre a válasz az, hogy a nagyobb vállalatok még mindig a jóval költségesebb klaszterekben látják a nagy számolási teljesítményt, ám ez az ismereteik bővülésével változni fog. A játékok töredéke használja csak fizikai számításokra a GPU-t, így lehet, hogy az NVIDIA egy – a termék piaci élettartama alatt – kihasználatlan lehetőségre áldozott jelentős erőforrásokat.

Köszönetnyilvánítás

Köszönet témavezetőmnek, Dr. Tornai Róbertnek,
aki segítségével rendelkezésemre állt
a nap 24 órájában, a hét minden egyes napján.

Köszönet Dr. Tőkési Károlynak,
aki jelen mű megírását motiválta.

Irodalomjegyzék

Asm Shader Reference, 2007: *Microsoft Developer Network*
http://msdn2.microsoft.com/en-us/library/bb219840.aspx

Cg PL, 2007: *Wikipedia.org*
http://en.wikipedia.org/wiki/Cg_%28programming_language%29

CUDA, 2010: *NVIDIA*
http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf

CUDA, 2007: *NVIDIA*
http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf

CUDA Performance, 2008: *anandtech.com*
http://www.anandtech.com/print/2586

Device Types, 2007: *Microsoft Developer Network.*
http://msdn2.microsoft.com/en-us/library/bb219625.aspx

DirectX 8.1 Pipeline, 2007*: gamedev.net*
http://www.gamedev.net/columns/hardcore/dxshader3/

GeForce8800GTX, 2007: *prohardver.hu*
http://prohardver.hu/teszt/Geforce_8800_gtx_forradalmi_nagyagyu/nyomtatobarat/teljes.html

Geforce 8800 GTX review, 2006: *xtreview.com*
http://xtreview.com/review159.htm

GLSL PL, 2007: *wikipedia.org*
http://en.wikipedia.org/wiki/GLSL

John D. Owens, 2005: *A Survey of General-Purpose Computation on Graphics*
társszerzők: David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell, kiadó:EUROGRAPHICS

Lévay, 2008/1: *Párhuzamos PET rekonstrukciós algoritmusok implementálása*
TDK dolgozat, Debrecen, Debreceni Egyetem, 2008

Lévay, 2008/2: *Molekuladinamikai szimulációs szoftver fejlesztése elosztott rendszerekre*
TDK dolgozat, Debrecen, Debreceni Egyetem, 2008

Model4, 2007: *Microsoft Developer Network*
http://msdn2.microsoft.com/en-us/library/bb509657.aspx

Predicate, 2007: *Microsoft Developer Network*
http://msdn2.microsoft.com/en-us/library/bb147357.aspx

Rutherford, 2003: *Petar Maksimovic, Johns Hopkins University*
http://www.pha.jhu.edu/~c173_608/rutherford/rutherford.html

Shaders, 2007: *Wikipedia.org*
http://en.wikipedia.org/wiki/High_Level_Shader_Language

Szőke 2005: *Képfeldolgozás a DirectX® 9 magas szintű árnyaló nyelvének segítségével*
szerző: Szőke Imre
TDK dolgozat, Budapest, Budapesti Műszaki Főiskola, 2005

The Geforce 6 Series GPU, 2005: *GPU Gems 2*
szerző: Randima Fernando Emmett Kilgariff
oldalszám: 475-485
kiadó: NVIDIA Corporation