Szakdolgozat

Dandár Gábor

Debrecen

2008

Debreceni Egyetem
Informatikai Kar

A shader nyelvek lehetőségeiről
A GPU felhasználása általános célú számításokra

Témavezető:
Dr. Tornai Róbert
Egyetemi adjunktus

Készítette:
Dandár Gábor
Programozó
Matematikus

Debrecen
2008

Tartalomjegyzék

Bevezetés

A mai videokártyák grafikus processzorai (GPU) rendkívül hatékony, és rugalmas erőforrások. Ez megmutatkozik programozhatóságukban, pontosságukban és teljesítményeikben egyaránt. Magas szintű programozási nyelvek (shader nyelvek) segítségével, e hatékony számítási erőforrások elérhetővé válnak számunkra.

A GPGPU nem más, mint „General Purpose Computation on GPUs”, azaz általános célú számítások a GPU-n. Kutatók észrevették, hogy egyes nem grafikai problémák megoldása nagyságrendekkel felgyorsítható, ha a CPU helyett a GPU-t használjuk.

Egy konkrét CPU és grafikus kártya teljesítményének összehasonlításakor a következő adatokat tapasztalhatjuk:

- *3.0 GHz Intel Core2 Duo:*

  Számítás: 48 GFLOPS

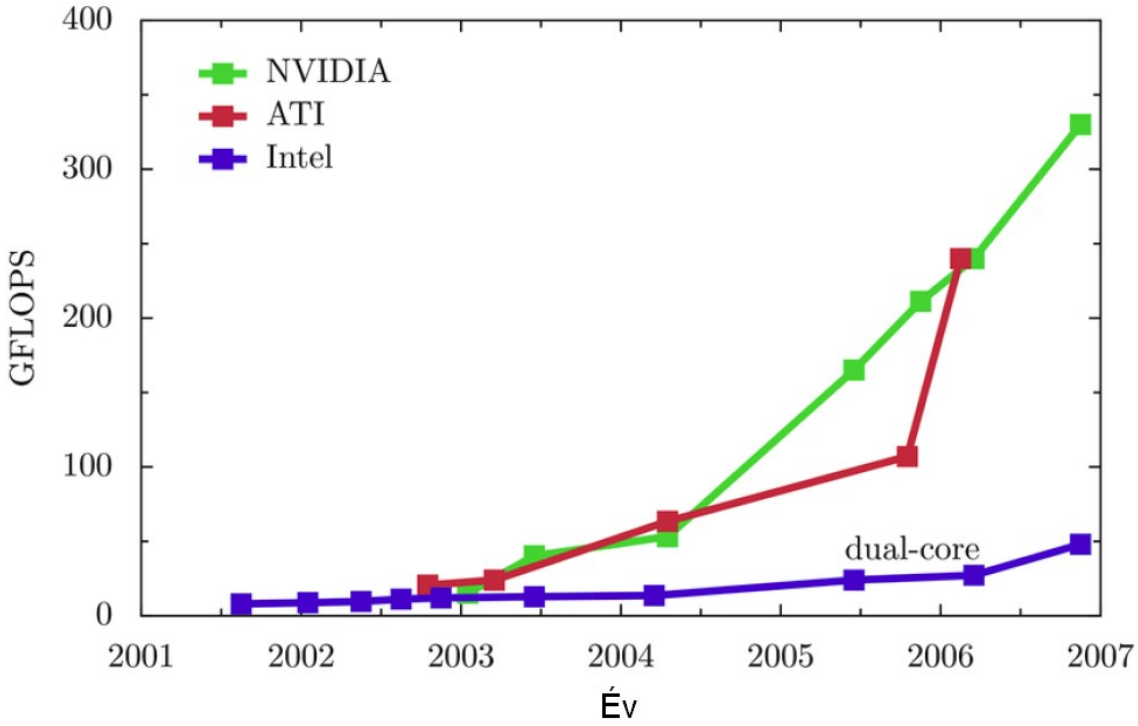
  Memória sávszélesség: 21 GB/s

- *NVIDIA GeForce 8800 GTX:*

  Számítás: 330 GFLOPS

  Memória sávszélesség: 55.2 GB/s

A FLOPS a másodpercenként elvégezhető lebegőpontos műveletek számának a mértékegysége (**FL**oating point **OP**erations **P**er **S**econd, GFLOPS = gigaFLOPS **)**. A grafikus processzorok egyre gyorsabbak, és gyorsabbak, amit remekül szemléltet a hagyományos processzorokkal összevetett sebességnövekedésük is (1. ábra).



1. ábra: A GPU-k és CPU-k sebességnövekedésének összehasonlítása

Mindezek mellett, amennyiben szeretnénk kihasználni a grafikus hardver olcsó erőforrásait, számos akadályba ütközhetünk. Ezek a kártyák elsősorban a videójáték piac számára készülnek, így fejlődésüket is ez befolyásolja a legnagyobb mértékben. A programozási modell szokatlan, és a rendelkezésre álló erőforrások is korlátozottak. Ahhoz, hogy hatékony GPGPU programokat készítsünk, járatosnak kell lennünk a számítógépi grafikában, valamint annak számítási kifejezésmódjaiban, és még ekkor is számos buktatóval vagyunk körülvéve.

Dolgozatomban szeretném megmutatni, hogy a mai grafikus kártyák erőforrásait kihasználva, egyes általános célú számításaink hatékonysága jelentősen növelhető. Ismertetem a dolgozathoz készített programot, ami egy egyszerű példája annak, hogy miként lehetséges nem grafikai célú számításokat végezni a grafikus processzoron, valamint bemutatnám az NVIDIA CUDA technológiáját is, ami jelentős változásokat hozott a GPGPU világba.

A GPU szemben a CPU-val

A mai nagy teljesítményű mikroprocesszorok az általános célú alkalmazások számára készülnek, melyek eltérő igényekkel rendelkeznek, mint a grafikus csővezeték (graphics pipeline). Ezek az alkalmazások jobban igénylik az összetett vezérlést mint a párhuzamosságot, és kevésbé vannak kiélezve a teljesítményre, mint a grafikus megjelenítést igénylő társaik. Egy CPU és egy GPU tervezésénél különböző célok játszanak szerepet, melynek eredményeképpen a grafikus csővezeték számára a CPU nem egy jó választás, ahogyan más, hasonló karakterisztikával rendelkező alkalmazás számára sem.

A CPU-k programozási modellje általában soros, melyek alkalmazásaikban nem mutatnak adat párhuzamosságot. Adat párhuzamosság alatt azt értjük, mikor több feldolgozó egység ugyanazt a műveletet hajtja végre különböző adatokon, ugyanabban az időpillanatban. A CPU programozási modelljében egy időpillanatban leggyakrabban egy adat feldolgozása folyik, és nem használ adat párhuzamosságot. A CPU-k utasításkészletének kibővítésével (pl. Intel SSE), illetve a többmagos processzorok megjelenésével elérhetővé vált az adat párhuzamosság ezeken a hardvereken is, de összehasonlítva a GPU-k nyújtotta párhuzamossággal, ez még mindig csekély.

A legfőbb oka annak, hogy a párhuzamosságot biztosító hardverelemek kevésbé elterjedtek a CPU-kon, az az, hogy a tervezők több tranzisztort biztosítottak a vezérlő hardvernek. Egy CPU-n futó programnak sokkal összetettebb vezérlésre van szüksége, mint egy GPU-n futónak, így nem véletlen, hogy a CPU tranzisztorainak nagy hányada a komplex vezérlést megvalósító funkcionalitásokat implementálja. Következésképpen egy CPU paneljének csak egy kisebb töredéke foglalkozik magával a számításokkal.

Mivel a CPU-k az általános célú programok számára készülnek, ezért nem tartalmaznak speciális funkciókat ellátó hardverelemeket. Ezzel szemben a GPU-k tartalmaznak speciális célú hardvert egyéni feladatok ellátásához, ami sokkal nagyobb hatékonyságot eredményez, mint amit az általános célú programozási megoldás valaha is nyújthatna.

Legvégül, a CPU memória rendszere a lehető legkisebb késleltetésre van optimalizálva, nem úgy mint a GPU, ahol a cél a maximális átbocsátás elérése. A CPU programoknak olyan gyorsan kell visszaadniuk a memória referenciákat, amilyen gyorsan csak lehet. Ennek érdekében a CPU-k memória rendszerei különböző szintű cache-elést tesznek elérhetővé ennek a késleltetésnek a minimalizálására. Azonban a cache használata többféle adattípus tekintetében is hatástalan, beleértve a grafikus inputokat, illetve azon adatokat melyek elérésére csak egyszer van szükség. A grafikus csővezeték esetében az egyes elemek késleltetésének minimalizálása helyett, az összes elem átbocsátásának a maximalizálása a memória rendszer hatékonyabb kihasználását, és nagyobb teljesítmény elérését eredményezi.

Stream alapú számítások

Ebben a fejezetben bemutatásra kerül a *stream* (adatfolyam) programozási modell, amely olyan módon strukturálja a programokat, hogy azok hatékonyak legyenek a számítások és a kommunikáció terén egyaránt. Ez a programozási modell az alapja napjaink GPU programozásának is.

A stream programozási modell

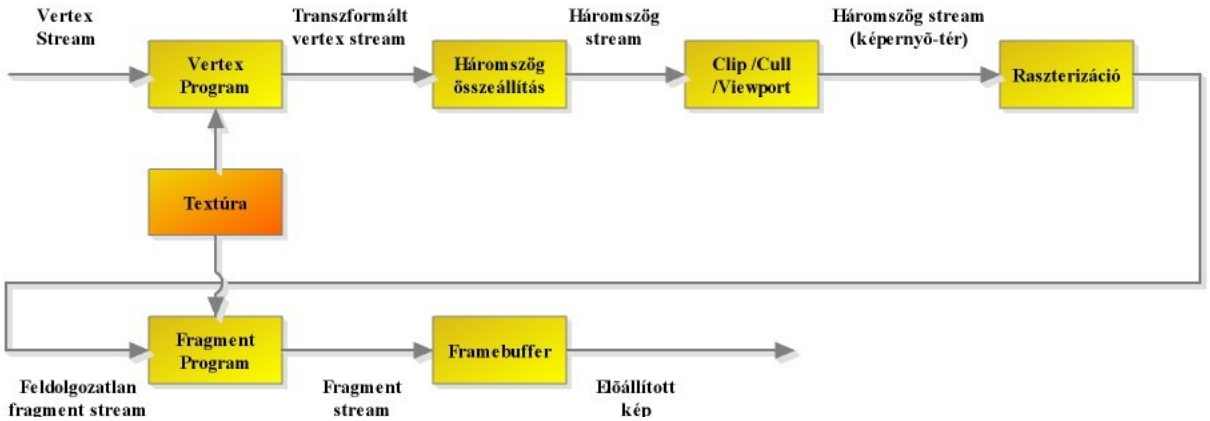
A stream programozási modellben minden adat streamként van reprezentálva, ami nem más, mint azonos típusú adatok rendezett halmaza. Az adattípus lehet egyszerű (egész, lebegőpontos...), vagy összetett (háromszögek, transzformációs mátrixok). A streamek hossza tetszőleges lehet, de észre fogjuk venni, hogy a streameken végzett műveletek akkor a leghatékonyabbak, amikor a stream hosszú (több száz elemet tartalmaz). A stream műveletekhez hozzátartozik a másolás, al-stream képzés, indexelés különböző index streamekkel, illetve számítások elvégzése a streamen *kernelek* segítségével.

Egy kernel teljes streameken működik. Inputnak egy vagy több streamet kaphat, és outputnak egy vagy több streamet produkálhat. A fő jellemzője a kerneleknek, hogy nem egyéni elemeken operálnak, hanem teljes streameken. A kerneleket leggyakrabban arra használjuk, hogy egy input stream minden egyes elemére végrehajtsuk ugyan azt a műveletet. Például egy transzformációs kernel pontoknak egy streamjét leképezi egy másik koordinátarendszerbe. A kernel műveletek közé tartozik többek között a bővítés (minden egyes input elemhez több output elem jön létre), a redukció (több input elem kombinációjából áll össze egyetlen output elem), és a filterezés (az output az input stream egy részhalmaza).

A kernel outputja mindig a kernel inputjának a függvénye, és a kernelen belül egy elemen elvégzett számítás soha nem függhet egy másik elemen elvégzett számítástól. Ezen megszorításoknak két fő előnyük van. Először is a kernel végrehajtásához szükséges adat teljesen ismert már a kernel írásakor (vagy fordításakor). Így a kernelek hatékonysága növelhető azzal, hogy a kernel inputokat és a köztes számítási eredményeket lokálisan tároljuk. Másodszor, a különböző stream elemeken megkövetelt független végrehajtás egyszerűvé teszi a kernel műveletek leképezését adat-párhuzamos hardverekre.

A stream programozási modellben az alkalmazások több kernel összeláncolásával állnak elő. Például ahhoz, hogy a grafikus csővezetéket implementáljuk a stream programozási modellben, írnunk kell egy vertex program kernelt, egy háromszög összeállító kernelt, egy a vágásokat elvégző (clipping) kernelt, és így tovább (2. ábra). Ez a modell világosan mutatja a kernelek közötti kommunikációt.

A grafikus csővezeték remekül illeszkedik a stream programozási modellhez. Különböző számítási állomásokra van bontva, melyeket adatfolyamok kapcsolnak össze egymással. Ez a struktúra megfelel a stream programozási modell kernel és stream absztrakcióinak. Az egyes állomásoknál előállított adat azonnal felhasználásra kerül a következő állomás által. Ez megegyezik a stream programozási modell kerneleinek a viselkedésmódjával.



2. ábra: A grafikus csővezeték a stream programozási modellnek megfelelően

Hatékony számítás

A stream modell több módon teszi lehetővé a hatékony számítást. A legfontosabb, hogy az alkalmazásaikban a streamek párhuzamosságot mutatnak. Mivel a kernelek teljes streamekkel dolgoznak, ezért a stream egyes elemei feldolgozhatóak párhuzamosan adat-párhuzamos hardver segítségével. Több elemből álló, nagyobb méretű streamek ezt az adat szintű parallelizmust magasabb fokon tudják kihasználni. Egyetlen elem feldolgozásakor utasítás szintű parallelizmust valósíthatunk meg. És mivel az alkalmazások több kernelből épülnek fel, ezért a kernelek láncolhatóak, és feldolgozhatóak párhuzamosan feladat szintű parallelizmus segítségével.

Az alkalmazás felosztása kernelekre megengedi speciális hardverimplementációk létrehozását egy, vagy több kernel végrehajtásához. Így ezek a speciális célú hardverek a nagyfokú hatékonyságukkal megfelelően alkalmazhatóak ebben a programozási modellben

Hatékony kommunikáció

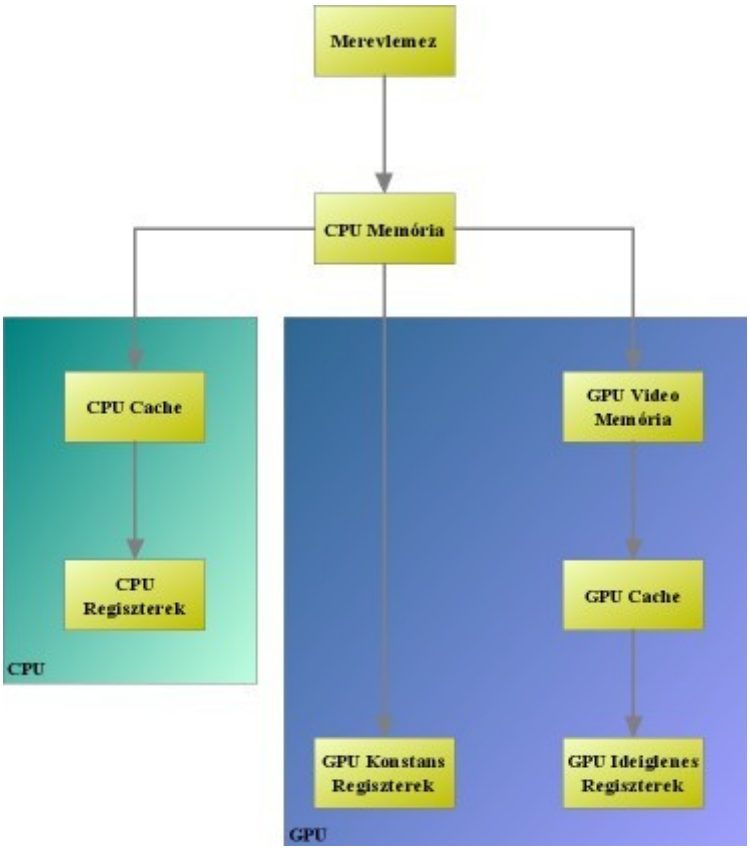
A hatékony kommunikáció ugyancsak az egyik elsődleges célja a stream programozási modellnek. Először is a kommunikáció sokkal hatékonyabb, ha teljes streameket mozgatunk a memóriába, vagy abból ki, mivel a transzfer előkészítéséhez szükséges költség egy teljes stream esetében csökkenthető, az elemek egyenkénti mozgatásakor viszont nem. Másrészről a kernel-láncokra strukturált alkalmazások megengedik a kernelek közötti köztes adatok on-chip módú tárolását, így azokat nem kell folyton a memóriából ki-be mozgatni. A hatékony kernelek képesek az inputjaikat és a köztes adatokat lokálisan a kernel végrehajtó egységen belül tárolni, emiatt az adat referenciák nem kerülnek le a chipről, vagy mennek azon keresztül a cache-be, ami a CPU-k esetében gyakran előfordul. Legvégül a végrehajtás csővezetékesítése megengedi a hardvernek, hogy hasznos munkát végezzen mindaddig, míg az adat vissza nem kerül a globális memóriából. A késleltetés tűrésnek ez a foka megengedi továbbá a hardverek átbocsátóképességre optimalizálását is.

A GPU memóriamodellje

A grafikus processzorok saját memória hierarchiával rendelkeznek, amely hasonlít a hagyományos soros mikroprocesszorokéhoz (tartalmazza a fő memóriát, cache-t, és a regisztereket). Ez a hierarchia grafikus műveletek felgyorsítását szolgálja, és leginkább a stream programozási modellhez illeszkedik. Ezen kívül a grafikus API-k mint az OpenGL és a Direct3D csak speciális grafikus primitívek (vertexek, textúrák, frame bufferek) számára engedik meg a memória használatát. A fejezet áttekintést ad a mai GPU-k memóriamodelljéről, illetve bemutatja, hogy hogyan illeszkednek ehhez a stream alapú számítások.

A memória hierarchia

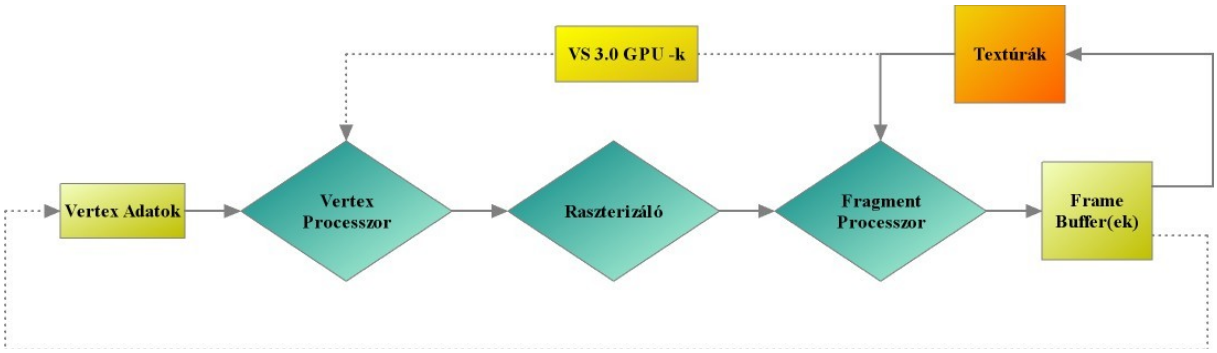
A grafikus processzorok memóriarendszere egy külön elágazást képez a mai modern számítógépek memória hierarchiájában (3. ábra). A CPU-hoz hasonlóan a GPU-ban is megtalálhatók a cache-ek és regiszterek, melyek a számítások alatti adathozzáférést gyorsítják. A GPU-k is rendelkeznek saját memóriával, ami azt jelenti, hogy a programozónak a program végrehajtása előtt explicit módon át kell mozgatnia az adatokat a GPU-ra. Ez a művelet az alkalmazásoknak egy tipikus szűk keresztmetszete, de a PCI Express busz szabványának köszönhetően a közös CPU-GPU memóriahasználat a közeljövőben még inkább elérhetővé válhat.



3. ábra: A CPU és a GPU memória hierarchiája

GPU stream típusok

A GPU memóriájának használatát illetően több megszorítás is van, és hozzáférni is csak a grafikus programozási interfész absztrakcióin keresztül lehet. Ezek az absztrakciók mindegyike egy stream típusnak tekinthető, mely saját hozzáférési szabályokkal rendelkezik. A GPU programozása során háromféle stream típushoz férhetünk hozzá. Ezek a vertex, a textúra és a frame buffer streamek. A negyedik típus a fragment stream, amely teljes egészében a GPU-n belül jön létre, és dolgozódik fel. A 4. ábra egy modern GPU-hoz tartozó csővezetéket ábrázol a három elérhető streammel, illetve mely pontokon férhetünk ezekhez hozzá.



4. ábra: Streamek a GPU-n

A legújabb DirectX 10-et támogató kártyák esetében, a fenti ábra a vertex processzort követően egy úgynevezett „geometry” processzorral egészül ki, melynek működését ebben a fejezetben nem tárgyaljuk.

Vertex streamek

A vertex streameket a grafikus API-kban vertex bufferekként ismerjük. Ezek a streamek tartalmazzák a vertexek pozícióit, és egyéb attribútumait. Ezeket az attribútumokat általában a textúra koordinátákhoz, színekhez, normálvektorokhoz használjuk, de elhelyezhetünk bennük különböző, a vertex program számára szükséges inputokat is.

Egészen a közelmúltig a vertex streameket csak úgy tudtuk módosítani, hogy az adatokat a CPU felől a GPU-hoz irányítottuk, a GPU nem volt képes a vertex streameket írására. Ezt ma a „copy-to-vertex-buffer”, vagy a „render-to-vertex-buffer” technikák valamelyikével valósíthatjuk meg. Az első esetben a renderelés eredménye a frame bufferből a vertex bufferbe másolódik, míg a másodiknál közvetlenül a vertex bufferbe renderelünk. Így a GPU által írható vertex streamek segítségével ciklusokba szervezhetjük a stream eredményeket a csővezeték eleje és vége között.

Fragment streamek

A fragment streameket a raszterizáló állítja elő, és a fragment processzor dolgozza fel. Ezek a fragment program stream inputjai, de a programozó által közvetlenül nem elérhetők, mivel teljes egészében a grafikus processzor állítja elő és dolgozza fel őket. A fragment stream tartalmazza a vertex processzor által előállított összes értéket (pozíció, szín, textúra koordináták, stb...). Akárcsak a vertexenkénti attribútumok, a fragmentenkénti attribútumok is használhatóak napjainkban a fragment program számára szükséges tetszőleges stream értékek tárolására.

A fragment programok nem férhetnek hozzá véletlenszerűen a fragment streamekhez. Ha ezt megengednénk, akkor függőségek alakulnának ki a stream egyes elemei között, ami ellentmondana az adat-párhuzamos programozási modellnek. Ha egy algoritmus számára mégis a véletlenszerű hozzáférés szükséges, akkor a streamet először le kell menteni a memóriába, majd textúra streammé konvertálni.

Frame-Buffer streamek

A frame-buffer streameket a fragment processzor írhatja. Ezek a streamek általánosan a képernyőn megjelenítendő pixeleket tartalmazzák. Ugyanakkor a GPU számítások a köztes számítási adataikat tárolhatják frame-bufferekben, illetve az újabb GPU-k képesek több frame-bufferbe írni, ugyanabban az időpillanatban.

A frame-bufferek a vertex és fragment programok számára nem elérhetőek, csakis a CPU-val írhatóak, illetve olvashatóak a grafikus API-n keresztül.

Textúra streamek

A textúra az egyetlen olyan GPU „memória” amely a fragment programok, és a Vertex Shader 3.0-át támogató GPU-k vertex programjai által random módon hozzáférhetőek. Ha a programozónak szüksége van random módon indexelhető vertex, fragment vagy frame-buffer streamre, akkor azt először textúrává kell alakítania. A textúrákat a CPU, és a GPU is tudja írni, és olvasni. A GPU oly módon képes írni a textúrákat, hogy a frame-buffer helyett közvetlenül a textúrára renderel, vagy a frame-bufferből átmásolja az adatokat a textúra memóriába.

A textúrák 1, 2 illetve 3 dimenziós streamekként lehetnek definiálva, 1, 2, 3 dimenziós címzésmódokkal.

A GPU memória hozzáférése

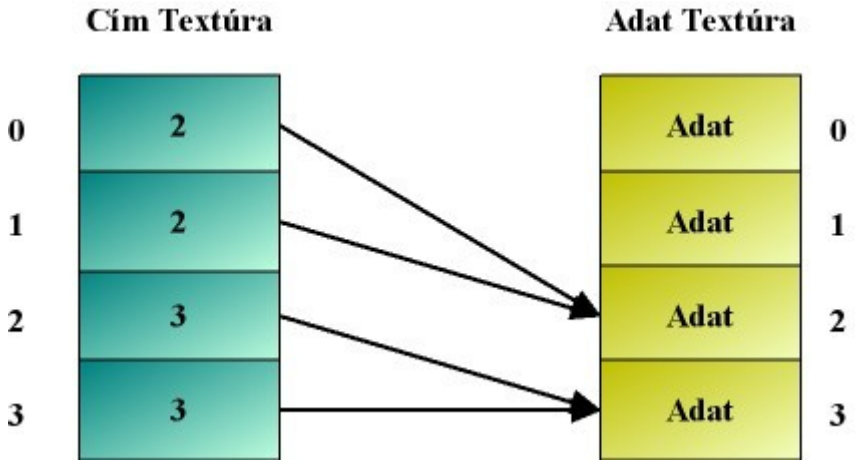
A vertex és fragment programok képezik a modern GPU-k motorját. A vertex programok a vertex streameken hajtják végre a műveleteiket, és az eredményt a raszterizáló felé továbbítják. A fragment programok fragment streameket használnak fel, és a frame-bufferbe írnak. Ezen programok képességei a végrehajtható aritmetikai műveletektől, és a hozzáférhető memóriától függnek. A GPU-n rendelkezésre álló aritmetikai műveletek száma közelít a CPU-kon felhasználhatókéval, viszont több memória hozzáférési megszorítás van jelen. Ezen megszorítások többsége a parallelizmus biztosítását szolgálják, melyek segítségével a GPU kihasználhatja a sebességi adottságait. Habár más megszorítások a GPU-k építőköveinek tekinthetőek, elképzelhető, hogy a következő generációkból kimaradnak.

Az alábbi lista a vertex és fragment kernelekre vonatkozó memória hozzáférési szabályokat tartalmazza azon GPU-kra vonatkozóan, melyek támogatják a Vertex Shader 3.0 és Pixel Shader 3.0 szabványokat.

- Nincs CPU memória és lemezhozzáférés
- Nincs GPU verem vagy heap
- Random hozzáférés a globális textúra memóriához
- Konstans regiszterek olvasása
    - A vertex programok használhatják a konstans regiszterek relatív indexelését
- Ideiglenes regiszterek írása/olvasása
    - A regiszterek a feldolgozott stream elemekre nézve lokálisak

- A regiszterek relatív indexelése nem megengedett
- Streamelt olvasás a stream input regiszterekből
  - A vertex kernelek a vertex streamekből olvasnak
  - A fragment kernelek a fragment streamekből (a raszterizáló kimenete) olvasnak
- Streamelt írás (csak a kernel végén)
  - Az írás helye az elem streambeli pozíciója által határozott
  - A vertex kernelek a vertex output streamekre írnak
  - A fragment kernelek a frame-buffer streamekbe írnak

Létezik egy további hozzáférési módszer is ami a fenti felsorolásból, és a korábban tárgyalt stream típusokból is kimaradt. Ez a *pointer stream*. A pointer streamek azt a lehetőséget használják ki, hogy bármely input stream felhasználható címként a textúra olvasáshoz (5. ábra). Ha egy pointer stream egy textúrából olvas, akkor azt *alárendelt textúrázás*-nak (*dependent texturing*) hívjuk.



5. ábra: Pointer Stream textúrával

Számítási fogalmak leképezése a GPU-ra

Ebben a fejezetben szeretném megmutatni, hogy hogyan tudjuk a szokásos számítási fogalmainkat leképezni a speciális sajátosságokkal rendelkező GPU-ra.

GPU-ra illeszkedő számítások

Az világos, hogy a GPU a legjobban a számítógépi grafika problémáinak a megoldásában a legjobb. A számítógépes grafikai számítások két legfőbb eleme az adat-parallelizmus és a függetlenség, azaz egy elemen elvégzett számítást csak kevéssé, vagy egyáltalán nem befolyásol egy másik elemen elvégzett művelet.

Aritmetikai intenzitás

Ez a két tulajdonság egyetlen fogalomban egyesül amit *aritmetikai intenzitás*-nak nevezünk, ami nem más, mint a számítás és a sávszélesség hányadosa:

aritmetikai intenzitás = műveletek száma / megmozgatott adatmennyiség

A mikroprocesszorokon elvégzett számítások költsége gyorsabban csökken mint a kommunikációé. Ez különösen igaz az olyan párhuzamos működésű processzorokra, mint a GPU. Mivel a technológiai újítások több tranzisztort tesznek lehetővé, ezért ezek közül több fordítható a funkcionális egységekre (pl.: aritmetikai logikai egységek) melyek a számítási teljesítményt növelik, mint a memória hierarchiára (cache), amellyel a memóriaolvasást gyorsíthatjuk.

Az előzőekből következően a legjobban a magas aritmetikai intenzitással rendelkező számítások képesek kihasználni a GPU-t. Egy nagyon jó példa ennek a szemléltetésére a lineáris egyenletrendszerek megoldása. Ezek a számítások jól működnek a GPU-n, mivel magasan adat-párhuzamosak. Nagy méretű adat-streamekből állnak (mátrixok és vektorok formájában) amelyekre ugyanazon számítási kerneleket kell alkalmazni. Az eredmény egyes elemeinek a kiszámításához kevés kommunikáció szükséges. Magas aritmetikai intenzitást igényel, és a memóriából átmozgatott adatok mennyisége alacsonyan tartható.

Stream kommunikáció: Gather & Scatter

A magas aritmetikai intenzitás eléréséhez a stream elemei közti kommunikációt a minimálisra kell csökkenteni. Mikor a GPU-n történő adatkommunikációról beszélünk, érdemes megemlíteni a kommunikáció két meghatározó fajtáját. Ezek a *gather* és *scatter* műveletek. A gather (gyűjtés) akkor hajtódik végre, mikor egy stream elemet feldolgozó kernel információt kér a stream többi elemétől, vagyis adatot „gyűjt” a memória más részeiről. A scatter (szórás) pedig akkor megy végbe, mikor a stream elemet feldolgozó kernel a stream többi elemének szolgáltat információt, azaz adatokat „szór” a memória egyéb területeire. Általánosan, a gather művelethez véletlenszerű memóriaolvasás, a scatter művelethez pedig véletlenszerű memória írás szükségeltetik.

A GPU számítási erőforrásai

Mielőtt rátérnénk a az általános számításaink GPU-ra történő leképezésére, ismerjük meg,

hogy milyen számítási erőforrásokat tud nekünk nyújtani a GPU.

Programozható párhuzamos processzorok

A GPU kétféle programozható processzorral rendelkezik. Ezek a vertex és fragment processzorok. A vertex processzorok vertex streameken hajtják végre a műveleteiket. A vertexek pozíció, szín, normálvektor és egyéb attribútumokból állnak, és a poligonok alapjait képezik. A vertex processzor egy *vertex programot (vertex shader*) hajt végre, amely *fragment stream* outputot állít elő, amit a fragment processzor egy *fragment program* (*fragment shader*) segítségével dolgoz fel, és állítja elő az egyes pixelek végleges színét.

Vertex processzor

A vertex processzor az inputként kapott vertexek pozícióját képes megváltoztatni. A mostani vertexprocesszorok többsége csak az éppen feldolgozás alatt álló vertex információihoz képesek hozzáférni. Ha belegondolunk, akkor egy vertex pozíciója meghatározza, hogy egy kép egyes pixelei hova rajzolódjanak ki. Egy kép nem más, mint egy tömb a memóriában ahova adatokat kell írni. Ebből kifolyólag a vertex processzor képes a scatter művelet végrehajtására.

Fragment processzor

A modern GPU-k több fragment processzorral rendelkeznek, és akárcsak a vertex processzorok, ezek is teljes mértékben programozhatóak. A fragment processzor párhuzamosan dolgozza fel a beérkező négy elemű vektorokat. Képesek továbbá textúrákból adatokat olvasni, azaz alkalmasak a gather művelet végrehajtására.

A GPGPU alkalmazások a fragment processzort sokkal erőteljesebben használják mint a vertex processzort. Ennek két fő oka van. Először is egy tipikus GPU sokkal több fragment processzort tartalmaz, másrészről az outputja sokkal inkább megy közvetlenül a memóriába. A vertex processzor kimenete először áthalad a raszterizálón, majd a fragment processzoron mielőtt elérné a memóriát.

Raszterizáló

Miután a vertex processzor transzformálta a vertexeket, ezen vertexek hármas csoportjaiból háromszögek képződnek, és ebből a háromszög streamből lesznek a fragmentek generálva. A fragmentek generálását a *raszterizáló* végzi el.

Textúrák

A fragment processzorok (és a Shader Model 3.0-át támogató vertex processzorok) képesek hozzáférni a memóriához textúrák formájában. Így gondolhatunk a textúrákra mint csak olvasható memória-interfészekre is.

Render-to-Texture

Mikor a GPU egy képet generál, akkor azt írhatja a frame-buffer memóriába a

megjelenítéshez, vagy írhatja a textúra memóriába. Az utóbbit nevezzük a *render-to-texture* funkcionalitásnak, ami a GPGPU részéről rendkívül meghatározó, mivel a segítségével közvetlenül visszacsatolhatjuk a GPU outputját az inputra.

Adattípusok

A hagyományos CPU-k programozásakor többféle adattípus áll rendelkezésünkre úgy, mint az integer, float és boolean. Ezen a téren a jelenlegi GPU-k sokkal korlátozottabbak. Habár némely magas szintű shader nyelv megengedi az integer és boolean adattípusok használatát, a jelenlegi GPU-k csak valós számokat dolgoznak fel fix, vagy lebegőpontos formában.

CPU-GPU megfeleltetések

Még egy tapasztalt CPU programozónak is nehézkes lehet a GPU programozása némi grafikus programozási ismeret nélkül. Az alábbi lista nagyon egyszerű megfeleltetéseket tartalmaz a hagyományos CPU fogalmak és azok GPU-n lévő megfelelői között.

- Streamek: GPU textúrák = CPU tömbök
- Kernelek: GPU fragment programok = CPU ciklusok
- Render-to-Texture = visszacsatolás
- Textúra koordináták = értelemezési tartomány
- Vertex koordináták = értékkészlet

Audió adatok feldolgozása a GPU-n

Ez a fejezet a dolgozat mellé beadott programhoz kapcsolódik, amely egy egyszerű effektet alkalmaz egy beolvasott audió fájlon a grafikus processzor segítségével. Ez egy jó példa arra, hogy hogyan alkalmazhatjuk a GPGPU-t a gyakorlatban adott grafikus API (jelen esetben OpenGL) segítségével.

Az audiófeldolgozás problémái

Az audió jelfeldolgozás széles körben van használva az otthoni felhasználókon át a stúdiókig. A feldolgozás során csatornánként különböző magas minőségű effektek kerülnek alkalmazásra. Minden egyes csatorna 44.1 és 192 KHz közötti sztereó audió jeleket tartalmaz, melyek mintánként (sample) 16 vagy 32 bitesek. Ez azt jelenti, hogy effektenként és csatornánként közel 1MB/s sebességű feldolgozás szükséges. A modern effektek összetettek, és alkalmazásuk hatékonysága nagy mértékben függ a processzortól, ami a szükséges számításokat végzi. Különösen fontos ez a real-time effekteknél, ahol a késleltetést érdemes 5-10 ms alatt tartani. Ideális esetben ezeket a számításokat speciálisan erre a célra kifejlesztett hardverek végzik, amikkel az a probléma, hogy igen drágák, így széles körben nem hozzáférhetőek.

A GPU mint lehetséges megoldás

Egy audió jelsorozat feldolgozásakor az adatok adatfolyamokon (streameken) keresztül érkeznek, melyek egyes elemein ugyanazon műveleteket kell végrehajtani. Könnyen észrevehető a hasonlóság a grafikus alkalmazásokkal, ahol szintén streameken operálunk. Felhasználva a GPU architektúráját és az azon megvalósítható adat-parallelizmust, az audió jelek ugyanúgy feldolgozhatóak párhuzamosan, mint a pixel adatok, amivel jelentős sebességnövekedést érhetünk el a CPU-val szemben.

A GPUAudio program

A program egy egyszerű példa arra, hogy milyen módon használjuk fel a GPU-t audió jelek feldolgozására. A futás során egy tömörítetlen WAV fájlból beolvassa az audió adatokat, melyeket a GPU számára értelmezhető textúrává konvertál, amit egy fragment program kap meg inputként.

A WAV fájl beolvasásához, és a lejátszáshoz az SDL (Simple DirectMedia Layer) multimédia könyvtárat használtam. Az audió adatok feldolgozása az OpenGL grafikus API-n keresztül történik OpenGL Shading Language (GLSL) nyelven megírt shader program segítségével. A render-to-texture funkcionalitást pedig az OpenGL FrameBuffer Object segítségével lett megvalósítva.

6. ábra: A GPUAudio program működés közben

Az audió adat beolvasása és konvertálása

Az audió adatok beolvasása egy *Wave* nevű struktúrába történik, ami tartalmazza az audió jeleket, illetve a lejátszáshoz szükséges egyéb attribútumokat.

```
typedef struct {
      SDL_AudioSpec spec;     /*SDL audio specification
                                structure*/
      Uint8   *sound;         /* Pointer to wave data */
      Uint32   soundlen;      /* Length of the wave */
      int      soundpos;      /* Current position in the wave*/
      } Wave;
```

A beolvasással egy időben megtörténik az SDL inicializálása, és beállításra kerül a *samplerate* (másodpercenként lejátszott minták száma), és a *bitrate* (egy minta mérete bitekben).

```
//Loads the wav file, and initializes SDL.
void sound::initSound(Wave &sample,int &samplerate, int &bitrate,
                  int &channels,
                  void(*callback)(void *unused, Uint8 *stream, int len))
{
      SDL_Init(SDL_INIT_AUDIO);
      SDL_LoadWAV("sample.wav",&sample.spec,&sample.sound,
```

```
                &sample.soundlen);
    printf("%i samples decoded\n", sample.soundlen);
    switch(sample.spec.format)
    {
        case AUDIO_U8:
        case AUDIO_S8:
            bitrate = 8;
            break;

        case AUDIO_U16LSB:
        case AUDIO_S16LSB:
        case AUDIO_U16MSB:
        case AUDIO_S16MSB:
            bitrate = 16;
            break;
    }

    samplerate = sample.spec.freq;
    channels = sample.spec.channels;
    sample.spec.size = sample.spec.samples * (bitrate/2);
    sample.spec.callback = callback;
    printf("bits per sample: %i sample rate: %i channels: %i\n",
          bitrate, samplerate, channels);
}
```

Az audió adatokat a GPU felé textúra formájában kell továbbítani. Az eredetileg beolvasott adatok előjel nélküli egészek, melyek egy egydimenziós tömbben tárolódnak. Ezt a tömböt kell kétdimenziós textúrává alakítani, amely lebegőpontos adatokat tartalmaz. A műveletet az alábbi függvény végzi el:

```
//Converts sound sample data to texture format
float* sound::soundToTexture(int texSize, int bitrate, int sampleCount,
Wave sample)
{
    float* texture = (float*)malloc(texSize * texSize * sizeof(float));
    short max = 0;
    if(bitrate == 8)
    {
        for(int i = 0; i < sampleCount; i++)
        {
```

```
                char val = ((char*)sample.sound)[i];
                if(val > max) max = val;
            }

            for(int i = 0; i < sampleCount; i++)
            {
                char val = ((char*)sample.sound)[i];

                // turn into unsigned and normalize to 0..1
                texture[i] = (val + 128.0) / (max + 128.0);
            }
        }
        else
        {
            for(int i = 0; i < sampleCount; i++)
            {
                short val = ((short*)sample.sound)[i];
                if(val > max) max = val;
            }

            for(int i = 0; i < sampleCount; i++)
            {
                short val = ((short*)sample.sound)[i];
                // turn into unsigned and normalize to 0..1
                texture[i] = (val + 65536.0) / (max + 65536.0);
            }
        }
        // zero empty samples
        for(int i = sampleCount; i < texSize * texSize; i++)
        {
            texture[i] = 0.0;
        }
        return texture;
}
```

Minden egyes sample egy előjeles nélküli egész, ami 8 bites sample méret esetén a [-128, 127] 16 bites esetén a [-32768, 32767] eshet. Ezeket az értékeket kell nekünk előjel nélküli lebegőpontos számokká konvertálni a [0, 1] tartományba. Ezt úgy tehetjük meg, hogy az

előjeles egész értéket előjel nélkülivé alakítjuk, majd elosztjuk a legnagyobb előjel nélküli értékkel. Ez a művelet sajnos igen költséges, és kerekítési hibák is keletkezhetnek.

Effekt számítása a GPU-n

A kapott adatokat a *glTexImage2D* függvénnyel rendeljük egy textúrához, amit a fragment programban használunk inputként:

```
uniform sampler2D DefaultTexture;
const float texSize = 512.0;

vec2 lookahead(vec2 coords, float index)
{
      float step  = 1.0 / (texSize- 1.0); //spaces between 0 and texSize
      float rowSize     = texSize * step;
      vec2 coords2 = coords;
      coords2.x = coords.x + index * step;
      if(coords2.x > 1.0)
      {
            float rowsUp = floor(coords2.x / rowSize);
            coords2.x = coords2.x - rowsUp * (1.0 + step);
            coords2.y = coords2.y + rowsUp * step;
      }
      return coords2;
}

void main()
{
      vec4 rColor = texture2D(DefaultTexture, gl_TexCoord[0].st).rgba;
      float s1 = texture2D(DefaultTexture, gl_TexCoord[0].st).r;
      s1 = s1 - 0.2;
      float s2 = texture2D(DefaultTexture, lookahead(gl_TexCoord[0].st,
                           50.0 * sin(gl_TexCoord[0].s))).r;
      s2 = s2 - 0.2;
      gl_FragColor = vec4(mix(s1,s2,0.5),0.0,0.0,1.0);
}
```

A fragment program a textúra adatain egy kórus effektet alkalmaz. Összemixeli az aktuális sample-t egy elkövetkezővel, aminek az indexét az aktuális sample szinuszából határozza meg. Ez kettő textúra lookupot, és egy lineáris interpolációt igényel.

A program outputja egy másik textúrába kerül render-to-texture alkalmazásával, amit az OpneGL FrameBuffer Object (FBO) segítségével valósítunk meg. Ez sokkal hatékonyabb és egyszerűbb módszer, mint a korábbi pbuffer megoldások. Az FBO mérete megegyezik a használt textúráéval, ami a kettő hatványai közül a legkisebb amibe még elfér az összes sample adat. Az FBO inicializálása, és a textúra méret meghatározása az *initFBO* függvénnyel történik:

```
//Initializes framebuffer object
void initFBO()
{
      glGetIntegerv(GL_MAX_TEXTURE_SIZE, &maxTexSize);
      printf("Maximum texture size: %i\n", maxTexSize);
```

```
	sampleCount = sample.soundlen;
	if(bitrate == 16) sampleCount /= 2;
	float root = sqrtf(sampleCount);
	int nextPowerOf2 = (int)ceilf(logf(root) / logf(2.0));
	texSize = (int)pow(2, nextPowerOf2);

	if(texSize > maxTexSize)
	{
		printf("sample size %i > max texture size %i\n",
			sample.soundlen, maxTexSize * maxTexSize);
		exit(1);
	}
	else
	{
		printf("sample size: %i  max texture size: %i\n",
			sample.soundlen, maxTexSize * maxTexSize);
		printf("using %ix%i texture size\n", texSize, texSize);
	}
	//Init FBO:
	glGenFramebuffersEXT(1,&fbo);
	glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,fbo);
}
```

Az outputként felhasznált textúrát az *initTextures* függvénnyel csatoljuk az FBO-hoz, valamint itt történik a konvertált adatok bemásolása egy másik textúrába, amit a korábban említett fragment program használ inputként.

```
//Initializes textures
void initTextures(void)
{

	glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
				GL_COLOR_ATTACHMENT0_EXT,
				GL_TEXTURE_2D,NULL,0);
	glDeleteTextures(1,&textureRead);
	glDeleteTextures(1,&textureWrite);

	//Init Render-to-Texture
```

```
glGenTextures(1, &textureWrite);
glBindTexture(GL_TEXTURE_2D, textureWrite);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8,  texSize, texSize,
            0, GL_RGBA, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

//attach to FBO:
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          GL_COLOR_ATTACHMENT0_EXT,
                          GL_TEXTURE_2D,textureWrite,0);

GLenum status = glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT);
if(status != GL_FRAMEBUFFER_COMPLETE_EXT)
{
      printf("FBO error!");
      exit(1);
}

glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

//the displayed texture
glGenTextures(1, &textureRead);
glBindTexture(GL_TEXTURE_2D, textureRead);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);

float* tmpSample = sound::soundToTexture(texSize, bitrate,
                                         sampleCount, sample);
printf("texSize: %d\n", texSize);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, texSize, texSize,
      0, GL_RED, GL_FLOAT, tmpSample);
```

```
      free(tmpSample);
    printf("init: %s\n", gluErrorString(glGetError()));
}
```

Az input textúra a vörös csatornájában (GL_RED) tárolja az adatokat, és a feldolgozott adatok is az output textúra vörös csatornájába íródnak.

Az effekt számítása a megjelenítés alatt történik. Ekkor a fragment program az input textúra adatait felhasználva az eredményt az output textúrába írja az FBO-n keresztül. Emellett megjelenítjük a program outputját is a képernyőn.

```
void drawScene()
{
      glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);

      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      glPushAttrib(GL_VIEWPORT_BIT);
      glViewport(0,0,texSize,texSize);

      glBindTexture(GL_TEXTURE_2D,textureRead);
      glEnable(GL_TEXTURE_2D);

      glBegin(GL_QUADS);
    {
        glTexCoord2f(0, 0); glVertex2f(-1, -1);
        glTexCoord2f(1, 0); glVertex2f( 1, -1);
        glTexCoord2f(1, 1); glVertex2f( 1,  1);
        glTexCoord2f(0, 1); glVertex2f(-1,  1);
    }
    glEnd();

      glPopAttrib();

      glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      glEnable(GL_TEXTURE_2D);
      glBindTexture(GL_TEXTURE_2D,textureWrite);
      glEnable(GL_TEXTURE_2D);
```

```
      glBegin(GL_QUADS);
    {
        glTexCoord2f(0, 0); glVertex2f(-1, -1);
        glTexCoord2f(1, 0); glVertex2f( 1, -1);
        glTexCoord2f(1, 1); glVertex2f( 1,  1);
        glTexCoord2f(0, 1); glVertex2f(-1,  1);
    }
    glEnd();
      glutSwapBuffers();
}
```

A kimenet konvertálása és lejátszás

Az adatokat a textúrából a *glTexImage* függvénnyel olvassuk vissza, ahol megadhatjuk, hogy milyen típusúként akarjuk azt visszanyerni (16 bites bitrate esetén GL_SHORT). Ezt előjelessé alakítva készen áll a lejátszásra.

```
void playSound(void)
{
            sample.sound = (Uint8*)malloc(sample.soundlen * sizeof(short)
            glBindTexture(GL_TEXTURE_2D, textureWrite);
            glGetTexImage(GL_TEXTURE_2D, 0, GL_RED, GL_SHORT,
                        sample.sound);
            sound::textureToSound16(sample.sound, sampleCount);
            sound::playSound(sample);
}
```

NVIDIA CUDA

Egészen mostanáig a GPU-ban rejlő teljesítmény kihasználása a nem grafikus alkalmazásokban egészen bonyolult volt:

- A GPU csak a grafikus API-n keresztül volt programozható amelynek megfelelő használata hosszú tanulási folyamatot igényel, és nem a legmegfelelőbb a nem grafikus alkalmazások számára
- A GPU DRAM-ja a hagyományos úton olvasható, azaz a memória bármely részéből tud olvasni információt (*gather*), viszont nem képes írni tetszőleges helyre (*scatter*) (7. ábra), amivel nagyon sokat veszít a rugalmasságából a CPU-hoz képest.
- Némely alkalmazásnak kritikus pontja a memória sávszélessége, csökkentve ezzel a GPU számítási teljesítményét.

Ez a fejezet egy újszerű hardver és programozási modellt mutat be, amely közvetlen megoldást nyújt a fent említett problémákra, és a GPU-t egy valódi adat-párhuzamos számítási eszközzé lépteti elő.

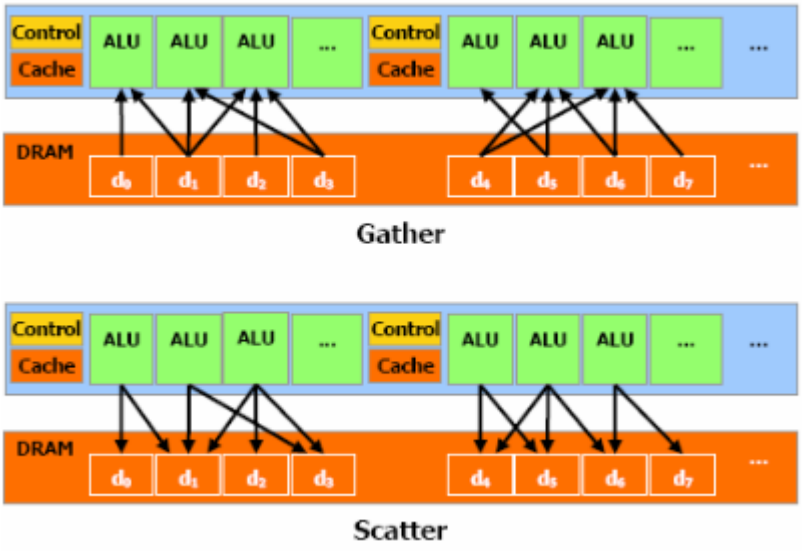
Új architektúra a GPU-n végzett számításokhoz

A CUDA jelentése *Compute Unified Device Architecture*, vagyis egy új hardver és szoftver architektúra amelynek célja, hogy ellássa és kezelje a GPU-n történő számításokat anélkül, hogy azokat le kellene képezni valamelyik grafikus API-ra. Ez elérhető a a GeForce 8-as szériákban (G8), a Tesla, és némely Quadro megoldásokban. A több CUDA és grafikus alkalmazás egyidejű futásakor a GPU-hoz való hozzáférés kezeléséért az operációs rendszer multitaszk mechanizmusa a felelős.

A CUDA szoftver verem több különböző rétegből áll. Egy hardver meghajtó, egy API és két magasabb szintű matematikai könyvtár általános használatra, a CUFFT és a CUBLAS. A hardver úgy lett tervezve, hogy támogasson egy könnyű driver, és futtató réteget a magasabb teljesítmény elérésének céljából.

A CUDA API tartalmazza a C programozási nyelv egy kiterjesztését is, ami rövid idő alatt egyszerűen megtanulható.

A CUDA általános memória címzést tesz lehetővé a nagyobb programozási rugalmasság elérésének céljából. Mind a gather és scatter műveletek támogatva vannak (7. ábra). Programozási szemszögből nézve, megvan a lehetőségünk arra, hogy írjunk, vagy olvassunk a memória tetszőleges területéről, akárcsak a CPU esetében.

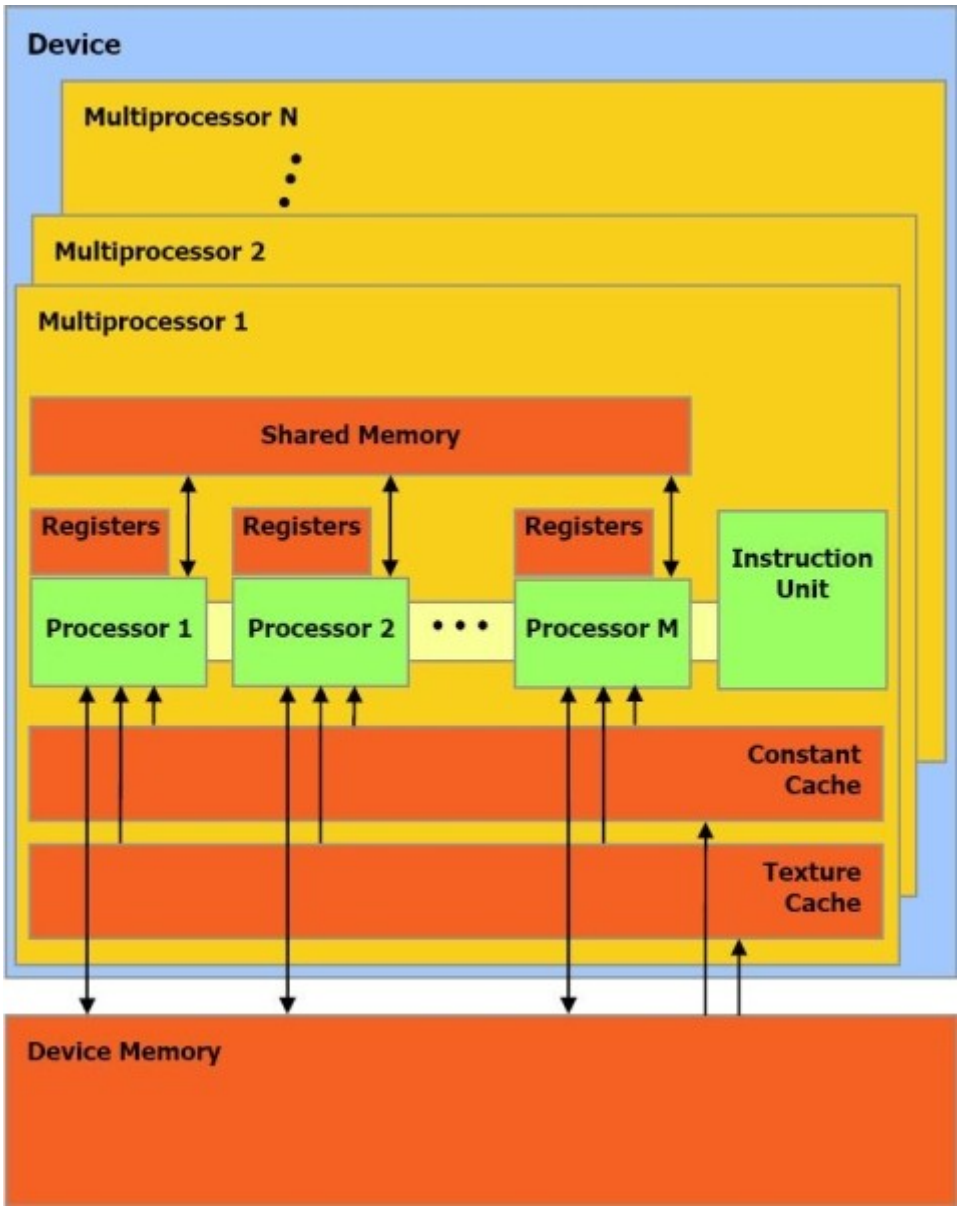


7. ábra: A Gather és Scatter memória műveletek

(NVIDIA CUDA Programming Guide)

A CUDA tartalmaz párhuzamos adat-cache-t (parallel data cache, vagy PDC), vagy más néven „megosztott memóriát” (shared memory), amely nagyon gyors írást és olvasást tesz lehetővé, és amellyel a szálak adatokat oszthatnak meg egymás között. Ez a megosztott memória az egyik kulcsfontosságú alkotóeleme a G80-as architektúrának, amely hatékonyabb lokális szinkronizációs alapokat nyújt.

Mielőtt továbbmennénk, érdemes ismertetni a az NVIDIA G8x és CUDA architektúráját (8. ábra).



8. ábra: Hardver modell

(NVIDIA CUDA Programming Guide)

Az eszköz több multiprocesszorból áll. Minden egyes multiprocesszornak megvan a maga SIMD architektúrája: Bármely órajel ciklus alatt a multiprocesszor minden egyes processzora ugyanazt az utasítást hajtja végre, de különböző adatokon operálnak.

Minden egyes multiprocesszor az alábbi négyféle on-chip memóriával rendelkezik:

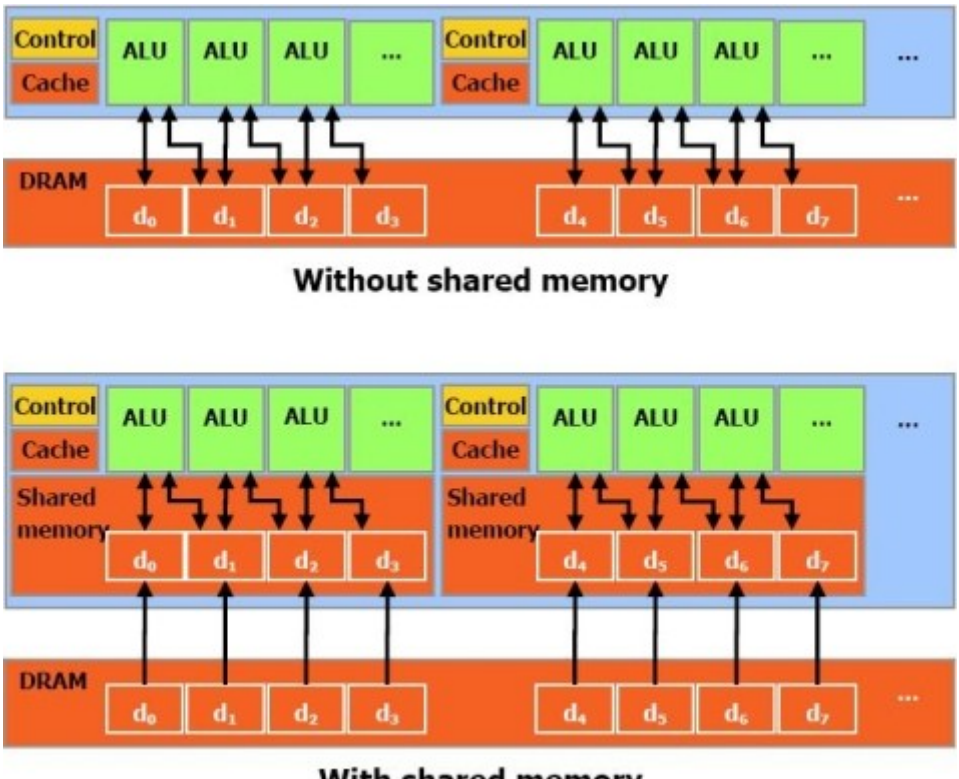
- processzoronként több 32 bites *regiszter*
- Egy PDC vagy *megosztott memória* amely az összes processzor között meg van osztva
- Egy csak olvasható *konstans cache* amely meg van osztva az összes processzor között, és a konstans memóriaterületről történő olvasást gyorsítja, amely az eszköz

memóriájának egy csak olvasható területeként van implementálva

- Egy csak olvasható *textúra cache*, amely meg van osztva az összes processzor között, és a textúra memóriaterületről történő olvasást gyorsítja, amely az eszköz memóriájának egy csak olvasható területeként van implementálva

A lokális és globális memóriaterületek írható és olvasható régióiként vannak implementálva a eszköz memóriájának, és nem cachelhetőek.

A fenti architektúra G80-as implementációjában multiprocesszoronként 8 processzor (ALU) található, és egy chipen 16 multiprocesszor helyezkedik el. Látható, hogy multiprocesszoronként egy megosztott memória van, és az eszköz memóriája mellet nincs lehetőség a különböző multiprocesszorok közötti kommunikációra. Röviden, nem létezik natív szinkronizációs módszer ennek egyszerűsítésére. Emiatt is szokás a G80 szinkronizációs funkcionalitását „lokális” jelzővel illetni, mivel nincs kiterjesztve a teljes chipre. Másrészről rendkívül hatékony megoldásokat tesz lehetővé számunkra.



9. ábra: Megosztott memória

(NVIDIA CUDA Programming Guide)

A 9. ábrán látható, hogy PDC-vel mindösszesen 4 memóriaolvasás szükséges a DRAM-ba, szemeben a korábbi GPGPU implementációk 6 olvasásához képest. Ez nagyobb hatékonyságot eredményez mint az automatikus cache, de több programozási trükkre is van szükségünk a megvalósításához.

A CUDA programozási modellje

A GPU CUDA-n keresztüli programozásakor egy olyan számítási eszköznek tekinthető, ami különösen nagy mennyiségű szálat tud végrehajtani párhuzamosan. Ekkor a CPU vagy *host* koprocesszoraként működik. Ez annyit jelent, hogy az alkalmazás azon részei melyek sokszor hajtódnak végre különböző egymástól független adatokon, azokat egy függvénybe különíthetjük, és a GPU-val végeztethetjük el a végrehajtásukat több különböző szálon. Ehhez a függvényt az eszköz utasításkészletének megfelelő programra (*kernel*) kell fordítani, majd feltölteni rá.

Szál batchelés

Egy kernelt végrehajtó szálak kötegei szál-blokkok rácsába vannak rendezve.

Szál blokk

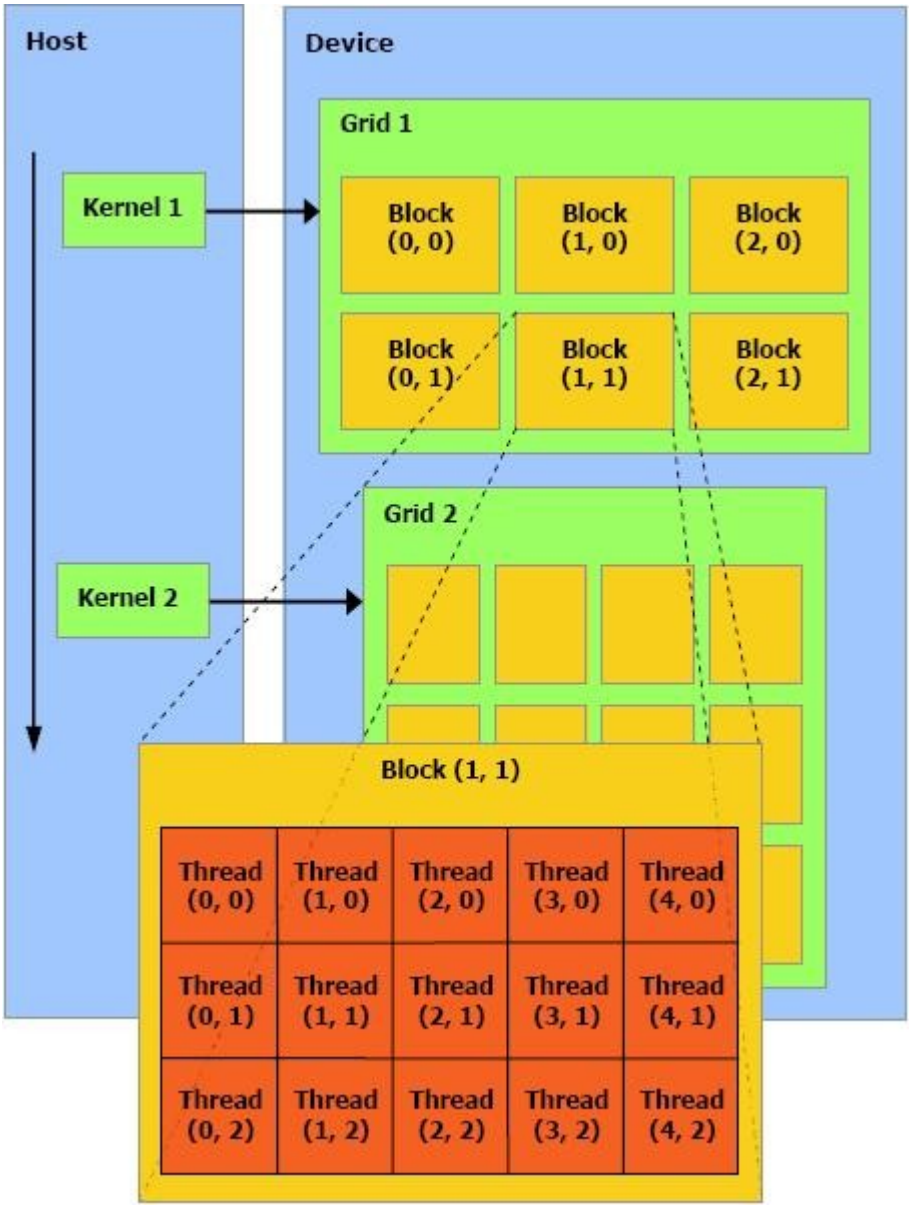
A szál blokk az szálaknak egy kötege, melyek az adatok megosztásával a gyors megosztott memórián keresztül működnek együtt, és szinkronizálják a végrehajtásukat a memória-hozzáférés koordinálásához. Még pontosabban, egy kernelen belül megadhatunk szinkronizációs pontokat, aholis a blokk szálai felfüggesztik a végrehajtásukat egészen addig, míg az összes szál el nem érte a szinkronizációs pontot.

Minden egyes szál egy *thread ID*-val van azonosítva, ami a szálnak a blokkon belüli sorszáma. A thread ID-n alapuló összetett címzés elősegítéséhez az alkalmazásban a blokkot meg lehet adni adott méretű 2 vagy 3 dimenziós tömbként is. Ekkor egy szálat 2 vagy 3 komponensű indexek segítségével azonosíthatunk. Egy kétdimenziós (D_x, D_y) méretű blokkban az (x, y) szál thread ID-ja $(x+yD_x)$, és egy (D_x, D_y, D_z) méretű háromdimenziós blokkban az (x, y, z) szál thread ID-ja $(x + yD_x + zD_xD_y)$.

Szál blokk rács

Az egy blokk által tartalmazható szálak maximális száma limitálva van, viszont az egyező méretű és dimenziójú blokkok melyek ugyanazt a kernelt hajtják végre, blokk rácsba kötegelhetőek. Így egyetlen kernel hívásakor sokkal több szál indítható. Ez viszont a szálak közötti együttműködés csökkenésével jár, mivel az egy rácsban elhelyezkedő, de különböző blokkban lévő szálak nem tudnak egymással kommunikálni, és a szinkronizációjuk sem lehetséges. Ez a modell megengedi, hogy a kernelek újrafordítás nélkül is hatékonyan fussanak különböző parallel lehetőségekkel rendelkező eszközökön. Ha kevesebb parallel képességgel bír, akkor a rács blokkjai futhatnak szekvenciálisan, ha pedig többel rendelkezik, akkor párhuzamosan, de gyakran előfordulhat ezen két lehetőség kombinációja is.

Minden egyes blokk egy *block ID*-val van azonosítva, amely a blokk rácson belüli sorszáma. A block ID-n alapuló összetett címzést elősegítve az alkalmazásunkban használhatunk adott méretű kétdimenziós rácsokat is, amelyben a blokkokat 2 komponensű indexek segítségével azonosíthatjuk. Egy (D_x, D_y) méretű kétdimenziós rácsban a (x, y) indexű blokk block ID-ja $(x+yD_x)$.



10. ábra: Szál batchelés

(NVIDIA CUDA Programming Guide)

Memória modell

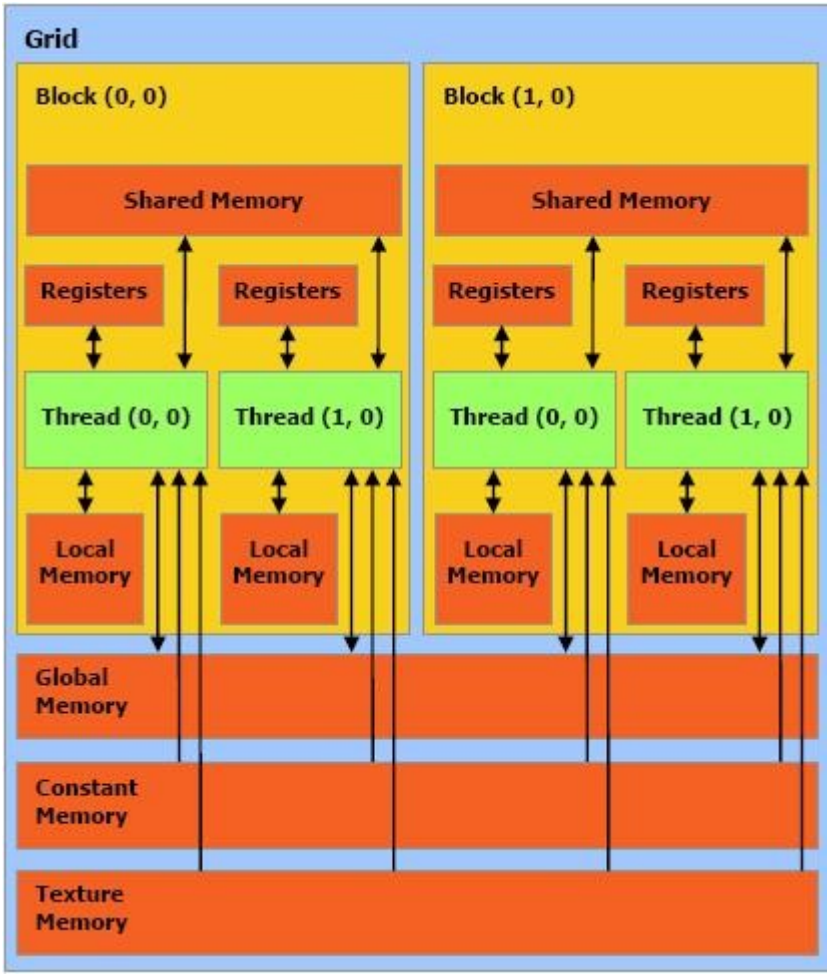
Egy futó szál az eszköz DRAM-jához és on-chip memóriájához az alábbi memóriaterületeken keresztül tud hozzáférni:

- Szálankénti írás/olvasás *regiszterek*
- Szálankénti írás/olvasás *lokális memória*
- Blokkonkénti írás/olvasás *megosztott memória*
- Rácsonkénti írás/olvasás *globális memória*
- Rácsonkénti olvasás *konstans memória*

- Rácsonkénti olvasás *textúra memória*

A globális, konstans és textúra memóriaterületek a host által írhatóak és olvashatóak, illeteve az egy alkalmazásból futtatott kernelek között perzisztens.

A globális, konstans és textúra memóriaterületek különböző memóriahasználatok szerint vannak optimalizálva. Továbbá a textúra memória különböző címzési módszereket nyújt, úgymint az adat filterezés speciális adatformátumokhoz.



11. ábra: Memória modell

(NVIDIA CUDA Programming Guide)

CUDA összefoglalva

A CUDA-ról összefoglalva az alábbiak mondhatók el:

- A CUDA az NVIDIA G80-as architektúráját egy az ANSI C-hez rendkívül közel álló nyelven és annak bővítményein keresztül világítja meg, és teszi elérhetővé a különböző GPU specifikus funkcionalitásokat

- A G80-nak 16 különálló multiprocesszora van, melyek egyenként 8 processzorból állnak amik az ütemező sebességének kétszeresével futnak. Minden egyes multiprocesszor 16KiB megosztott memóriával rendelkezik, és minden processzornak saját regiszter bankja van.
- A szálak blokkokba vannak csoportosítva. Az egy blokkban elhelyezkedő szálak ugyanazon multiprocesszoron történő futása biztosított, ezáltal kihasználhatják a megosztott memóriát, és a lokális szinkronizációt.
- A megosztott memória (parallel data cache vagy PDC) lehetővé teszi a programozónak, hogy csökkentse a szükséges DRAM hozzáférések számát az adatok többszöri újrafelhasználásával. Ha multiprocesszoronként több blokk is fut, akkor a megosztott memória csak egy bizonyos része érhető el az összes blokk számára.
- A lokális szinkronizáció rendkívül hatékony egészen addig, amíg egy multiprocesszoron több blokk fut. Továbbá a lokális memória teszi lehetővé a szinkronizáció közbeni kommunikációt.

Korlátok és a jövő

Jelenleg a dupla lebegőpontos (FP64) számítások nem támogatottak a mostani GPU-kon, emellett az alábbi megszorítások vannak jelen:

- A rekurzív függvények nincsenek támogatva. Ez az ütemező egy megszorítása, mivel a hardver oldalon nincsenek valódi függvények, de még ha lennének is, nincsen verem amelyben a push és pop műveletekkel mozgathatóak lennének az argumentumok.
- A GPU-n nincsen hatékony út a globális szinkronizációra, amely a kernel felosztását, és a CPU-n történő szinkronizálást vonja maga után. A számos multiprocesszort és egyéb tényezőket figyelembe véve, ez sem a tökéletes megoldása a problémának.
- Nincsenek változó argumentum számú függvények. A probléma ugyanaz, mint a rekurziónál.
- A lebegőpontos számok egésszé konvertálása eltérő módon megy végbe, mint az x86-os processzorokon.
- A sávszélesség és a késleltetés a GPU és CPU között szűke keresztmetszetet eredményezhet.

A fenti problémák többsége a jövőben feltehetően megoldódik, főleg a globális szinkronizáció területén remélhetünk előrelépéseket.

Összefoglalás

A fenti fejezetek tartalmának ismeretében nyugodtan jelenthetem ki, hogy a GPU ma már nem csak számítógépi grafika problémáinak megoldására alkalmas, hanem más számítási célokra is felhasználható. Egyik legnagyobb előnye a CPU-val szemben, hogy adat párhuzamosságot igénylő feladatok ellátása sokkal gyorsabban történik rajta. Ez többek között a processzoron lévő speciális hardverelemeknek köszönhető, melyek a CPU-k esetében nélkülözésre kerültek a minél általánosabb felhasználás érdekében.

Bemutattam a stream programozási modellt, ami a GPU-k programozásának az alapját képezi. Ez a modell jól illeszkedik a grafikus csővezeték felépítéséhez. A csővezeték minden egyes állomása egy streamet kap inputként, aminek a feldolgozása során ugyancsak egy stream áll elő, amit továbbít a következő állomásnak a csővezetékben. A GPU különböző stream típusai (vertex, fragment, frame-buffer és textúra streamek) eltérő szerepekkel bírnak a GPGPU alkalmazások esetében.

A dolgozathoz készített GPUAudio programom jól szemlélteti, hogy egy viszonylag egyszerű probléma megoldása is igen körülményessé válik, ha azt a grafikus processzor segítségével szeretnénk megoldani. A programozónak jártasnak kell lennie a számítógépi grafikában, valamint ismernie kell a grafikus API-k valamelyikét (DirectX, OpenGL). A hagyományos programozási fogalmainkat át kell ültetni a GPU-n fellelhető erőforrásokra, amiket nem minden esetben tudunk olyan módon használni, mint a CPU-n található megfelelőiket. Jó példa erre a scatter művelet. A fragment program outputjaként kapott stream elemeinek a memóriabeli helye már az input feldolgozása előtt meg van határozva. Ezen kívül több megszorításba ütközhetünk a memóriakezelés, a vezérlés és a felhasználható adattípusok területén is. Az imént említett problémák és megszorítások halmaza az idő haladtával egyre szűkül. Jó példa erre az NVIDIA által nemrégiben kiadott CUDA architektúra.

Úgy gondolom, hogy a CUDA megjelenése is bizonyítja, hogy a grafikus kártyák felhasználási területe ma már sokkal tovább mutat mint a számítógépi grafika. A GPU nyújtotta teljesítmény felhasználható tudományos célú számításokhoz, fizikai szimulációkhoz, vagy éppen a kriptográfiában. Ezekhez hasonló alkalmazás írásakor a korábbiakban nem volt lehetőség a grafikus API megkerülésére. A CUDA egy ANSI C alapú nyelvet biztosít, ami mellett nincsen szükségünk a DirectX vagy OpenGL használatára, amivel rengeteg terhet vesz le a programozó válláról. Ezenkívül a memóriakezelés területén fennálló problémák egy részére is megoldást nyújt. Fontos előrelépésnek tartom, hogy a memória bármely területéről képesek vagyunk olvasni, és az írás is bárhova történhet, azaz mind a gather mind a scatter műveletek támogatva vannak. A megosztott memória (PDC) használatával a memória-hozzáférések csökkenthetőek, továbbá a lokális szinkronizáció is növeli a hatékonyságot, és a kommunikáció területén is történtek előrelépések.

Véleményem szerint, egyes nagy aritmetikai intenzitást igénylő alkalmazások tervezésénél a GPU-t mint számítási erőforrást komolyan figyelembe lehet venni. Elnézve a sebességük jelenlegi növekedését, a jövőben még nagyobb teljesítményre számíthatunk, illetve további fejlődés várható a CUDA, vagy az ATI CTM technológiák területén is. Ezeket figyelembe véve úgy gondolom, hogy a grafikus processzorok felhasználási spektruma még tovább fog növekedni, és talán az általános célú alkalmazások szerves részévé válhat később.

Ábrajegyzék

Irodalomjegyzék

[1] Matt Pharr, Randima Fernando – GPU Gems 2. Addison-Wesley Professional 2005.

[2] Randi Rost – OpenGL Shading Language (2nd Edition). Addison-Wesley Professional 2006.

[3] NVIDIA CUDA Programming Guide (1.1)

[4] SDL API Reference Guide (1.2)

[4] http://www.gpgpu.org/s2007/ - SIGGRAPH 2007 GPGPU COURSE

[5] http://www.gamedev.net/reference/articles/article2331.asp – OpenGL Frame Buffer Object 101

[6] http://www.beyond3d.com/content/articles/12/1 - NVIDIA CUDA Introduction