DIPLOMAMUNKA

Farkas László

Debrecen

2011

DEBRECENI EGYETEM

INFORMATIKAI KAR

KOMPLEX KÉPFELDOLGOZÓ RENDSZER ALGORITMUSAINAK GPU TÁMOGATÁSA

Témavezető:

Dr. Zichar Marianna

Egyetemi Adjunktus

Készítette:

Farkas László

Programtervező Matematikus

Debrecen

2011

Tartalomjegyzék

1. Bevezetés

A GPGPU, azaz General-Purpose computing on Graphics Processing Units, napjaikban egyre hangsúlyosabb szerepet kap a nagy számítás igényű algoritmusok megvalósításánál. A korunk video kártyái nem csak grafikai számításokra alkalmasak, hanem általános célú programokat is tudunk futtatni rajtuk, amit annak köszönhetünk, hogy processzoraik fix- és lebegőpontos számításokat tudnak végrehajtani, melyeket tetszés szerint vezérelhetünk is.

A modern technológiának köszönhetően egyre több processzort tartalmaznak ezek a kártyák. Tekintve, hogy ezeknek a GPU processzoroknak az utasításkészlete jóval kisebb, mint egy hagyományos CPU-é, így kisebb méretűek, melynek köszönhetően adott területen jóval több fér el belőlük. Például napjaink egyik grafikus csúcs kártyája, az nVidia [1] Tesla C2070 [2], 448 ilyen processzort tartalmaz, és dupla pontosság esetén 515 Gigaflops, azaz másodpercenként 515 milliárd műveletre képes, miközben szimpla pontosságú lebegőpontos számítás esetén már Teraflops nagyságrendet tud elérni. Nem véletlen, hogy a jelenlegi legerősebb szuperszámítógépekben is jelen vannak a grafikus processzorok.

Dolgozatom célja, hogy megvizsgáljam ezen technológia segítségével milyen hatékonyság javulást lehet elérni a már meglévő képfeldolgozási algoritmusok tesztimplementációin. Ezek az implementációk a „DRSCREEN – A cukorbetegség szemszövődményeinek szűrésére alkalmas képfeldolgozó rendszer kifejlesztése” című projekt [3] keretein belül kerülnek megvalósításra. Amennyiben a hatékonyság javulás kellő mértékű, úgy egy új technológia kerülhet az ilyen területen érdekelt kutatók, illetve programozók kezébe. Természetesen ennek elengedhetetlen része az is, hogy miként is lehet megvalósítani olyan kódokat, melyek segítségével a grafikus kártya processzorait működésre tudjuk bírni.

2. A GPU technológia

A video kártya processzorain futtathatunk C nyelvű kódokat, melyeket ki kell egészíteni olyan eszköztárral, mely tartalmazza a GPU kezelő utasításit. Két ilyen könyvtárkészlet terjedt el a köztudatban. Az egyik az nVidia által létrehozott és preferált CUDA [4], a másik a Khronos [5] csoport által szabványosított heterogén programozási eszközrendszer, az OpenCL [6].

2.1. CUDA bemutatása

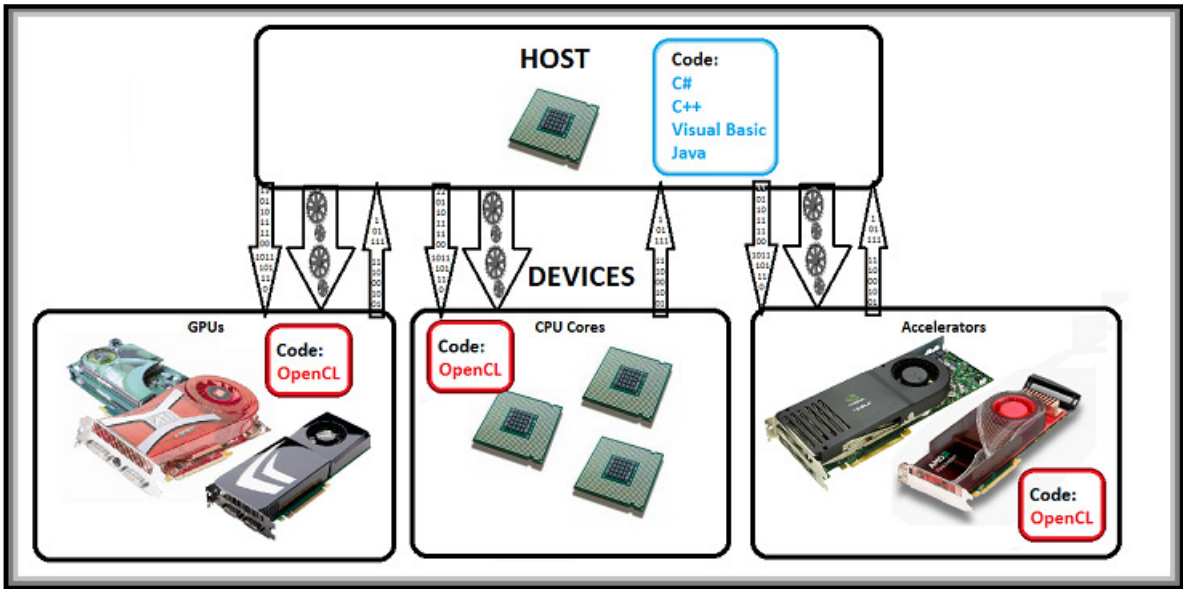
CUDA, azaz Compute Unified Device Architecture, az nVidia párhuzamos programozási rendszere, melynek segítségével az nVidia grafikus kártyáit tudjuk programozni.

2007-ben jelentette be az nVidia, hogy elkészítették ezt az APIt, melynek fő feladata az volt, hogy segítségével könnyen készíthessünk olyan kódokat, melyek képesek a GPU-n bizonyos tudományos számítások, vagy másképpen mondva nem grafikai számítások elvégzésére, melyeket addig csak körülményesen tudtak megvalósítani a programozók. Viszont mivel ekkor még nem beszélhetünk szabványosításról, ezért a kódok kizárólag a megfelelő nVidia által gyártott GPU-kon voltak képesek működni. Ez a probléma sajnos azóta is jelen van, így csak akkor célszerű ezt a megoldást alkalmazni, ha olyan kódot akarunk írni, ami csak nVidia kártyákkal ellátott rendszeren akarunk működtetni.

2.2. OpenCL bemutatása

Az OpenCL, azaz Open Computing Language, egy olyan standard, melynek segítségével, párhuzamosan programozhatunk heterogén rendszereket (*1. ábra*). Heterogén alatt azt kell érteni, hogy elviekben ugyanazon kód képes futni mind GPU-n, mind CPU-n, mind más számoló egységeken

mindenféle átalakítás nélkül, így lehetőségünk van arra, hogy gyakorlatilag, ügyes felosztással a gép összes számoló egységét egyszerre működésre bírjuk. A gyakorlatban ez sajnos még nem valósul meg teljesen, mivel nincs egy olyan univerzális illesztő program, amivel minden gyártó minden eszközét tudnánk kezelni. Mivel azonban ez a technológia nem olyan régi, így remélhetőleg a jövőben születik erre a problémára megoldás.



1. ábra: Heterogén programozási modell (forrás: [7])

Az OpenCL funkcionalitását tekintve ugyanazt a célt szolgálja, mint a CUDA, csak azzal a lényeges különbséggel, hogy ez már szabványként lett létrehozva, így a gyártóspecifikáltság nincs jelen.

2.3. A két rendszer összehasonlítása

A CUDA egyik nagy hátránya, hogy csak az nVidia eszközein lehet alkalmazni, szemben az OpenCL rendszerével, melyet minden megfelelő driverrel rendelkező eszközön le tudunk futtatni, igaz, ezen driverek egyszerre nem tudnak működni. Mivel az eszközök gyártói szállítják az implementációt az illesztő programjaikkal, és a Microsoft Windowson belül

csak egy `dll` fájlban lehet jelen, így két különböző gyártótól származó eszköz közül egyszerre csak az egyik működhet. Ez eléggé nagy probléma, de ennek ellenére még mindig hordozhatóbb rendszer, mint a CUDA.

A nVidia fejlesztéseit, új függvényeit egyből integrálja is saját rendszerébe, míg OpenCL-ben csak bizonyos idő elteltével lesznek ezek elérhetők. Emiatt a CUDA valamivel mindig naprakészebb rendszerként van jelen a piacon.

Ezeken felül a CUDA megoldásai egy-két helyen egyszerűbbnek, könnyebben megérthetőnek tűnnek, de funkcionalitásukat tekintve nincs lényegi különbség a két rendszer között.

Jelenlegi fejlesztéseimhez az OpenCL-re esett a választás, az alapján, hogy a későbbiekben hasznos lehet, hogy olyan függvényeket, eljárásokat hozzunk létre, amelyek akár egy nVidia, akár egy AMD [8], akár egy grafikus kártya nélküli gépen is tudnak futni.

A későbbiekben az OpenCL-en keresztül fogom bemutatni, hogy miként is lehet megvalósítani egy video kártyán futó programot, de azt végig szem előtt kell tartani, hogy a két rendszer között csak gyakorlati eltérések vannak - függvények neve, paraméterezése – az elvi szintű megvalósítás azonban ugyanaz.

3. Képfeldolgozási algoritmusok hatékonyságjavítása

A képfeldolgozó algoritmusok nagy általánosságban jellemezhetőek azzal, hogy viszonylag nagy adathalmazon nagy számítás igényű műveletek, így ebből kifolyólag viszonylag lassúak. A nagy adathalmazt maga a kép jelenti, mivel egy viszonylag kisebb kép is több mint egymillió képpontból áll és ezen felül több színcsatornával is rendelkezik. A nagy számítási igény, pedig úgy jelenik meg, hogy ezek az algoritmusok jellemzően minden egyes képpontra valamilyen műveletet végeznek. Például egy egyszerű konvolúció, mely egy 7×7-es maszkkal dolgozik, képpontonként 49 szorzást és 49 összeadást –ami egy fentebb említett kép esetén majdnem 300 millió műveletet– jelent. Szekvenciális futtatással ez az algoritmus, a műveletek számából adódóan, viszonylag csak lassan tud lefutni. Ebből adódóan lényeges szempont ezeknek az algoritmusoknak a felgyorsítása.

3.1. Párhuzamosíthatóság

A képfeldolgozó algoritmusoknál gyakran előfordul, hogy az eredményképnél az adott képpontok kiszámítása független a többitől, így azon műveletek egymástól függetlenül párhuzamosan is kiszámíthatóak.

A párhuzamosíthatóság kérdése viszont összetettebb megközelítést kíván.

Első lépésben azt kell megvizsgálni az adott algoritmusoknál, hogy azok milyen szinten párhuzamosíthatóak. A kérdés megválaszolására az algoritmusokat különböző hierarchikus (*2. ábra*) osztályokba sorolhatjuk.

Képi szint: Ebbe az osztályba eső algoritmusok párhuzamosíthatósága az egyes input képek függetlenségén alapul. Minden egyes kép külön szálon futhat, mivel eredményük egymástól független, de az eredmény meghatározására az egész képre szükség van.

Részképi szint: Az ide tartozó algoritmusok esetén, az eredmény függősége csak a bemeneti kép egy adott részétől függ. Ilyen algoritmusok például a jelöltállítók, ahol az egyes jelöltek meghatározásához azok környezetére is szükség van, de a különböző jelöltek egymástól függetlenül előállhatnak.

Pixel szintű: Azon algoritmusok tartoznak ebbe a csoportba, melyeknél az eredményképen az összes pixel a többi eredményeitől függetlenül meghatározható. Az előfeldolgozó algoritmusok jelentős részére ez a megszorítás teljesül. Például egy konvolúciónál az eredménykép összes képpontjának meghatározásához, csak az inputkép adataira van szükség és lényegtelen, hogy az eredménykép mely részei lettek már meghatározva.

2. ábra: Párhuzamosíthatósági szintek hierarchiája

Egy másik megközelítési szempont lehet, hogy egy összetettebb algoritmus esetén annak egyes részei, hogyan és milyen szinten párhuzamosíthatók. Például lehetne vizsgálni, hogy egy előfeldolgozás során a benne szereplő algoritmusok egymástól függetlenül, párhuzamosan végrehajthatóak-e. A GPU sajátosságaiból kiindulva, viszont ezen megközelítés a jelentőségét elveszti, mivel az egyes multiprocesszorok szálain ugyanazok az utasítások futnak le párhuzamosan, így ezeken fizikailag is kivitelezhetetlen, hogy különböző algoritmusok fussanak le, valamit az egyes multiprocesszorok eltérő programozhatóságára sincs eszközünk. Így

nem javasolt az az elképzelés, hogy különböző algoritmusokat párhuzamosan futtassunk a GPU-n, és jóformán lehetetlen is, továbbá könnyedén belefuthatnánk a memória limitáltságba. Emiatt célszerű csak egy-egy függvényt, vagy szekvenciálisan egymás után végrehajtható függvények sorát megvalósítani a grafikus kártyánkon.

3.2. GPU-ra optimalizálható kódok

A GPU-n történő számítás hatékony kihasználásához figyelembe kell venni a benne lévő lehetőségeket és a korlátozottságokat. A GPU processzoroknak a számítási teljesítményük kisebb egy hagyományos CPU maghoz képest, elemben nagy számban vannak jelen a grafikus kártyán, így célszerű olyan algoritmusokat választani GPU-n történő futtatásra, melyek képesek egyszerre meghajtani az össze processzorát. Ezért fontos szempont, hogy az adott algoritmus mennyire párhuzamosítható. Nyilvánvalóan egy pixel szintű párhuzamosíthatóságra képes kódot sokkal célszerűbb megvalósítani GPU-n mint egy olyat, melyet csak képi szinten lehet.

A GPU programozás során egy másik fontos szempontot is figyelembe kell venni, ami pedig a memória kihasználása. Tekintve a tényt, hogy egy GPU lényegesen kevesebb memóriával rendelkezik, mint maga a rendszer, így fontos azt is figyelembe venni, hogy a program ne lépje túl a rendelkezésre álló memória méretét. Igaz egy közepes video kártya is rendelkezik legalább fél gigabyte RAM-mal, de ebből nem áll az összes a rendelkezésünkre. A rendszer a többi háttérben futó programnak is fenntart egy bizonyos memóriaterületet, valamint a GPU-n majd futtatandó kernel kód is itt kerül tárolásra. Szerencsére a fennmaradó memória terület még így is elegendő akár több kép tárolására is, viszont olyan algoritmusok, melyek futásuk során több átmeneti képet is előállítanak, azoknál előfordulhat ez a probléma. Az ebből származó hibákat, csak úgy tudjuk elkerülni, ha a részeredmény képeket eldobjuk, vagy ha azok lényegesek, akkor vissza kell másolnunk azokat a gép hagyományos memóriájába. Ez a megoldás több olyan problémát is felvethet,

amelynek eredményeként a kódunk akár lassabb is lehet, mivel a CPU és a GPU között történő adatmozgatás viszonylag lassú.

A harmadik lényeges szempont pontosan ennek a problémának az elkerülésére vonatkozik. A lehető legkisebbre kell csökkenteni a CPU és a GPU memóriája közti adatátvitelt. Az algoritmusok megírásánál ennek is fontos szempontnak kell lennie. Ha van több olyan algoritmus mely ugyanazt a képet, vagy egymás eredményeit dolgozzák fel, akkor célszerű ezeket összevonni úgy, hogy csak a végeredmény kerüljön vissza a CPU-hoz, vagy más néven a host-hoz. Persze ez csak akkor lehetséges, ha a többi szempont nem sérül súlyos mértékben az összevonás által.

Ezek után megállapíthatjuk, hogy olyan algoritmusokat érdemes megvalósítani a grafikus kártyákon, melyeknél lehetőség szerint a globális eredmény meghatározható legyen sok részeredmény összevonásával, melyek viszonylag sok számítást követően egymástól függetlenül előállnak. Emellett lehetőség szerint kevés adatmozgatással járjanak a központi-, illetve a GPU memóriája között.

4. Algoritmusok tesztimplementációjának hatékonyságjavítása

A DRSCREEN projektben lévő képfeldolgozási algoritmusok hatékonyságjavítása előtt fel kellet mérni a már meglévő algoritmusok párhuzamosíthatósága, és GPU-n alkalmazhatósága mellett azt is, hogy milyen környezetben lettek azok implementálva. Ezek többsége MATLAB [9] alatt lett megvalósítva, mivel napjainkban ez a programozás rendszer kitűntetett figyelmet kap számos kutatási területen, így felmerült az a kérdés is, hogy MATLAB-on miként lehet grafikus kártyát meghajtó programokat létrehozni, illetve azok milyen hatékonysággal működnek. Mivel a MATLAB a GPU technológiának nem az alapértelmezett platformja, így ez a kérdés fontos szerepet kapott.

4.1. Jacket

Az AccelerEyes [10] cég foglalkozott ezzel a kérdéssel, és létrehoztak egy MATLAB alatt működő toolbox-ot, mely a Jacket [11] nevet kapta. Ennek filozófiájában komoly hangsúlyt kapott az egyszerűség. Ez többek között annak is köszönhető, hogy az egész MATLAB ezt a filozófiát követi, mely szerint a kutatások során fontos a gyors, egyszerű implementálhatóság, aminek köszönhetően több idő juthat a kutatás lényegi részére.

Az egyszerűség úgy mutatkozik meg, hogy lényegében bizonyos programozási elemek – változók típusai, függvények – nevei elé egy g betűt beillesztve máris egy GPU-n futó kódot kaphatunk.

```
A = zeros(5);
B = gzeros(5);
```

A fentebbi példán az `A` és a `B` változó egy 5×5 zéró-mátrix lesz, viszont amíg az `A` egy hagyományos a központi memóriában helyet kapó változó,

addig a `B` a GPU memóriájában foglalódik le. Ha két GPU-n lefoglalt változó között valamilyen műveletet végrehajtunk, az akkor a GPU-n fog lefutni, feltéve, ha az adott művelet implementálva van a toolbox-ban.

A párhuzamosíthatóság kérdését is viszonylag egyszerűen megoldották a `gfor` segítségével, mely az előírt lépésszámú ciklust, más szóval a `for` ciklusokat értelmezte újra.

```
B = gzeros(5);
i = [];
gfor i = 1:25
    B(i) = B(i)+1;
gend
```

A fenti példán a `B` mátrix minden elemét megnöveljük eggyel, viszont a szokásos iteratív módszer – azaz először a `B(1)`, majd a `B(2)` és így tovább – helyett az „iteráció” lépései párhuzamosan hajtja végre a rendszer. Figyelembe kell venni viszont, hogy ezt nem minden esetben lehet alkalmazni; például ha egy vektor `i.` eleméhez hozzá akarjuk adni az `i-1.`-et, akkor ez párhuzamosan teljesen más eredmény fog adni, mint iteratívan. Szekvenciálisan végrehajtva az utolsó elem a vektor elemeiknek összege lesz, párhuzamosan viszont csak az utolsó véletlen számú elem összege lesz.

A toolbox másik jelentős hibája, hogy jelenleg az alapfüggvényeket leszámítva alig van olyan függvény, ami implementálva van. Emiatt sok, a projektben használt függvényt nekünk kell megírni, ami az egyszerűség rovására megy, illetve a függvények megírásánál a limitáltság miatt olyan problémákba lehet ütközni, amelyet ezzel a rendszerrel kifejezetten nehéz megoldani.

4.2. MEX

Amennyiben az adott algoritmusokat mindenféleképpen át kell írni, akkor jogosan merül fel az a kérdés, hogy ez esetben akkor miért nem egyből egy OpenCL vagy CUDA kódot készítünk. Viszont mivel az algoritmusoknak csak egy része írható át GPU-n futó párhuzamos kódra, így a kódoknak a többi részét is át kellene írni C nyelvre. Vagy keresni kell egy olyan eszközt, mellyel a C kódokat integrálni tudjuk MATLAB környezetbe.

A MATLAB-nak van egy ilyen beépített rendszere melynek segítségével a C kódjainkat kisebb módosítások árán futtatni tudjuk a MATLAB rendszere alatt. Ez az eszköz a MEX [12], mely végül is egy C fordító, azzal a különbséggel, hogy ezzel egy bináris m fájllá tudjuk fordítani a kódjainkat, melyet később egy egyszerű függvényhívásként alkalmazni tudunk a MATLAB-ban. Ennek a módszernek így megvan az az előnye, hogy a GPU kódokat fokozatosan tudjuk beépíteni a rendszerbe, anélkül, hogy a kutatás folyamatossága ne sérüljön.

A MEX alkalmazásához 4 egyszerű lépést kell végrehajtani a C-kódban (*lásd: cd melléklet: GPUConv.cpp*):

1. Include direktívát alkalmazni kell a `mex.h`-ra, így elérhetőek lesznek a MEX Speciális függvényei.
2. Meg kell adni egy úgynevezett gateway függvényt, mely `mexFunction` névre hallgat.
3. Használnunk kell egy speciális struktúrát, `mxArray`-t, mely egyben reprezentálja a MATLAB összes lehetséges típusú tömbjét.
4. A MATLABból kapott, illetve oda szánt változók kezelésére speciális APIkat kell használni.

A gateway függvény teremti meg az átjárhatóságot, hogy a MATLAB-ból meg tudjuk hívni a C kódot. Ez a következőképpen néz ki:

```
void mexFunction(int nlhs, mxArray *plhs[], int
    nrhs, const mxArray *prhs[])
```

A `nrhs` a bemenő, azaz a MATLAB által átadott paraméterek száma, a `nlhs`, pedig a MATLAB által várt, kimenő paraméterek száma. A `prhs` és a `plhs` pedig a be-, illetve kimenő paraméterek egymásután egy tömbben. Ezeken a paramétereken keresztül tud kommunikálni a két kód.

Ha a szükséges módosításokat megvalósítottuk, akkor lefordíthatjuk a kódunkat, aminek eredményeképpen előáll az eredeti `c` vagy `cpp` fájllal megegyező nevű `mexw32` vagy `mexw64` kiterjesztésű fájl (*lásd: cd melléklet: GPUConv.mexw32*), attól függően, hogy hány bites platformara fordítjuk. Innentől kezdve a fájl nevét függvénynévként felhasználva a MATLAB kódunkon belül bármikor meghívhatjuk.

Ennek az eszköznek a segítségével egy OpenCL vagy CUDA kódot egy kisebb átalakítással fel tudunk használni MATLAB-ban, illetve ha ügyesen oldjuk meg a gateway függvény alkalmazását, akkor a függvényünket akár egy C nyelvű kódból is meg tudjuk hívni, vagy hasonló eszközzel akár egyéb programozási nyelvekben is fel tudjuk használni a későbbiek során.

5. Az OpenCL programozás főbb elemei

A hatékonyságjavításhoz fel kellet mérni azt, hogy valóban lehetséges-e a meglévő algoritmusoknál hatékonyabbakat előállítani a GPU technológia segítségével. Ennek a vizsgálatnak a leghatékonyabb módszere az, ha magunk készítünk egy grafikus kártyán futó kódot, és a futási idejét összehasonlítjuk az eredeti kód futási idejével.

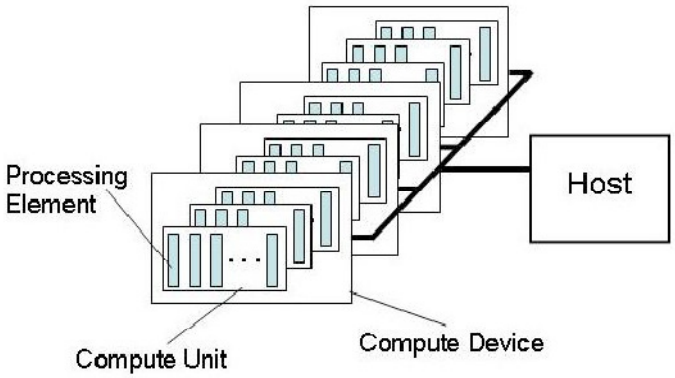
Ahhoz hogy egy GPU-n futó kódot létre tudjunk hozni elsősorban olyan vezérlő jellegű rutinokat kell alkalmazni, melyek segítségével a rendszert, vagy más szóval a platformot, fel tudjuk készíteni a GPU számítások elvégzésére. Ehhez viszont több dolgot is szemügyre kell venni, melyekkel megérthetjük miként is működik az OpenCL rendszere.

5.1. Platform modell

Az OpenCL, mint párhuzamos heterogén programozási rendszer egyik fontos feladata, hogy tudja kezelni az összes számoló egységet. Ezek kezelését a host látja el, az úgynevezett host kód segítségével. Ennek a C nyelven írt kódnak kell megvalósítani a platformot (*3. ábra*), azaz a teljes rendszer működtetését, az adatok továbbítását, illetve meg kell határozni az adatok felosztását, és még más dolgokat, melyeket később fogok kifejteni.

Egy rendszeren belül egy vagy több számítást végző eszköz van, melyet az angol szakirodalom device-nak nevez. Mivel heterogén rendszerről beszélünk, ezért az eszközök teljesen különbözőek lehetnek, így fontos a közös nyelv megléte. Ezt a problémát oldja meg az OpenCL a kernel kódok segítségével. A kernel kód végül is az a kódrészlet, amely majd az eszközökön fog futni. Ennek fő feladata a számítások elvégzése, az esetlegeses adatok mozgatása az adott eszközön, azaz a mi esetünkben a GPU-n belül, illetve a párhuzamos programozásból nélkülözhetetlen szinkronizációt is itt kell megvalósítani. Ezeket hogy meg tudja valósítani az eszközünk, a host-nak létre kell hozni

egy contex-et, mely tartalmazza az eszközök összességét, a kernel kódot, a kernel kódból létrehozott programot, illetve a memória objektumokat. Az eszközöket vezérlő utasítások pedig az úgynevezett parancssorban (command queue) helyezkednek majd el.



3. ábra: Az OpenCL platform modellje (forrás: [13])

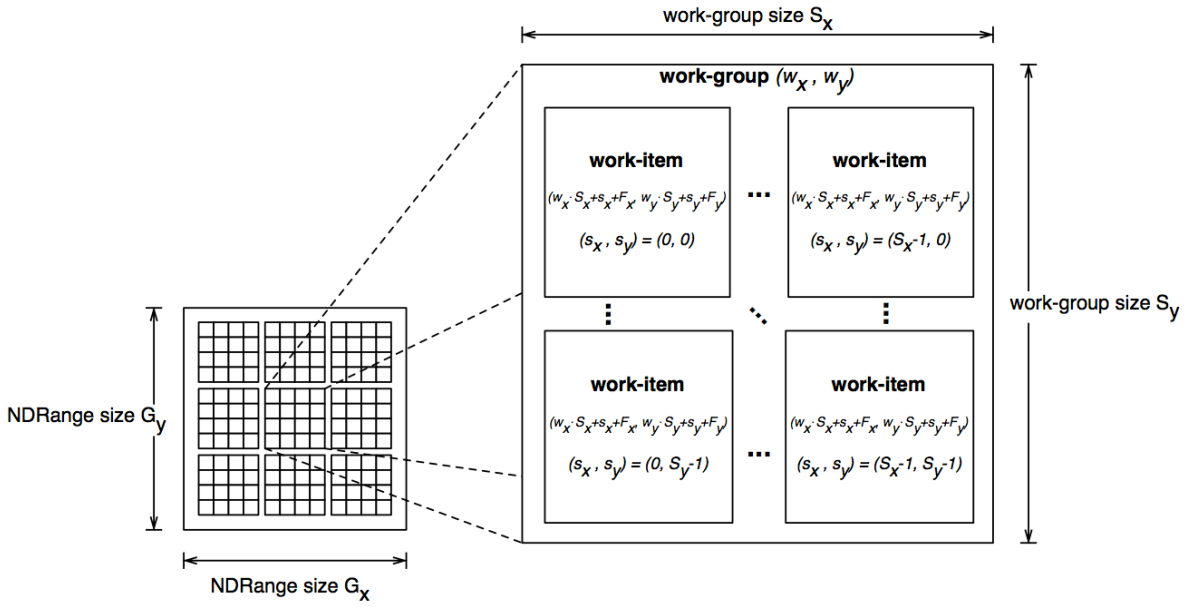
5.2. Végrehajtási modell

A számítások elvégzését végző egységek szerkezete kicsit eltér a megszokottól köszönhetően egyrészt a heterogén rendszernek, másrészt a számolóegységek eltérő számának.

A kernel kód utasításainak végrehajtásának legkisebb egysége a work-item, azaz munkaegység. A munkaegységek munkacsoportokba, más szóval work-group-ba csoportosulnak, illetve ezekből is több lehet jelen egy adott eszközön belül (*4. ábra*). A munkaegységeknek és a munkacsoportoknak is van saját egyedi ID-ja, melynek segítségével meg lehet határozni a programon belül betöltött globális pozícióját minden munkaegységnek. A programozás során technikailag minden egyes pixel, eredmény kiszámítását egy munkaegység fogja végezni.

Ezek a munkaegységek, és munkacsoportok nem csak 1 dimenziós, hanem két-, vagy háromdimenziós szerkezetben is reprezentálhatóak. Leggyakrabban

bináris képeken dolgozunk így a kétdimenziós ábrázolás fordul elő legtöbbször.

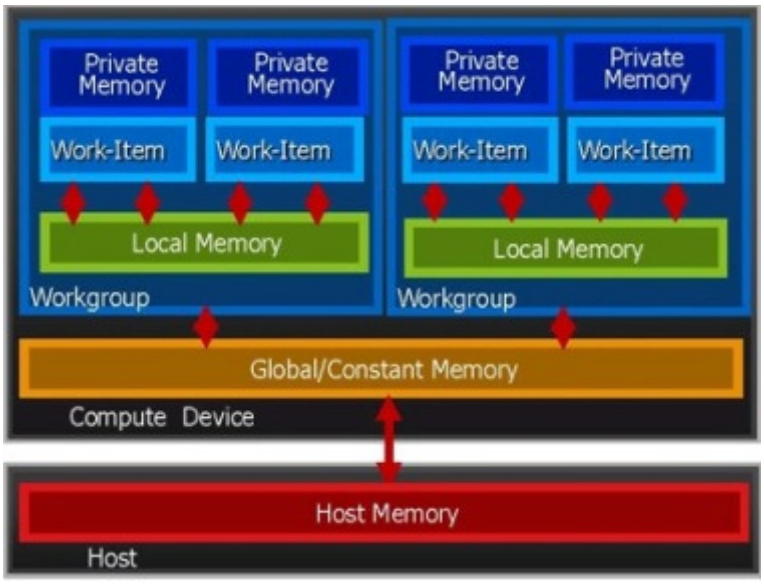


4. ábra: Az OpenCL végre hajtási modellje 2 dimenzióban reprezentálva (forrás: [13])

A modell ábrázolása azért ilyen, mert próbál alkalmazkodni a GPU-k architektúrájához. Rendszerint egy grafikus kártyán belül több multiprocesszort helyeznek el, amelyekben található számolóegységek száma sem egyenlő, így ezeknek a különbségeknek az elfedésére jött létre ez a modell. Minden egyes munkacsoport külön-külön multiprocesszoron fog lefutni, természetesen attól függően, hogy mennyi mag található az eszközön. A rendszer automatikusan szétosztja ezeket a munkacsoportokat. Ha például 8 munkacsoportot hozunk létre és 4 multiprocesszor található, akkor mindegyik 2 munkacsoportot fog feldolgozni, viszont ha csak 2 multiprocesszor van, akkor mindegyik négyet fog feldolgozni. Ugyanígy a munkaegységek pedig az adott multiprocesszor számolóegységeinek megfeleltethetőek lesznek szétosztva.

5.3. Memória modell

A GPU architektúra memóriamodelljén (*5. ábra*) négy különböző memória van megkülönböztetve, melyek között nem csak logikai, hanem fizikai különbségek is vannak. Logikailag fontos tényező, hogy milyen végrehajtási egység látja az adott memóriarészt. Fizikailag pedig az adott memóriarész elérési, írási, olvasási ideje között van eltérés. Ezen szempontokat kell figyelembe venni a programozás során is, ugyanis további jelentős gyorsulást érhetünk el, ha a gyorsabb elérésű memóriákat használjuk, amelyeket azonban kívülről nem tudunk elérni.



5. ábra: Az OpenCL memória modellje (forrás: [6])

- Host memória: Ez nem az architektúra része, viszont fontos szerepet játszik. Ez a központi memória esetünkben, ahonnan érkeznek a számításhoz szükséges adatok, illetve az eredményt is majd ide kell visszaküldeni. Ezt csak a host látja.
- Globális memória: Ez már a GPU-n található. Ezt az összes munkacsoport összes munkaegysége látja, tudja írni és olvasni, illetve ami még lényeges, hogy a host is ugyanúgy tudja írni, olvasni. Amikor a host-ról másolunk adatot az eszközre akkor az mindig ide fog kerülni.

- Konstans memória: A globális memória azon része mely a kernel futása során nem változik, csak a host tudja írni.
- Lokális memória: Ez már magán a GPU chipjén van rajta, így lényegesen gyorsabb az elérése, mint a globális memóriának. Minden munkacsoportnak külön-külön saját lokális memóriája van, amit csak az adott munkacsoportban található munkaegységek tudnak írni és olvasni. Ehhez a memóriához a host-nak már nincs hozzáférése.
- Privát memória: Szintén a chipen található, melyet csak az adott munkaegység láthat.

5.4. Host kód

A következő példán (*lásd: cd melléklet: con_orig.cpp*) keresztül szeretném bemutatni, hogy konkrétan milyen lépéseket kell végrehajtani egy host, később pedig egy kernel kódban, természetesen a részletes specifikáció [13] ismertetése nélkül. Jelen példámban az egyszerűség kedvéért egy konvolúciót fogok bemutatni, mely az érhálózatok detektálása [14] során került be a rendszerbe. Lényege az, hogy van egy maszkunk, amit ráillesztünk minden egyes képpontra, majd vesszük a maszk és az alatta lévő pixel értékének szorzatát és ennek összege lesz a középpontban lévő pixel új értéke.

Mindenekelőtt lehetővé kell tenni, hogy a program elérje az OpenCL speciális függvényeit, változóit, ezt az `#include <opencl.h>` paranccsal lehet megoldani.

Először is meg kell határozni a platformot, amin dolgozni szeretnénk.

```
cl_platform_id clPlatform;
clGetPlatformIDs(1, &clPlatform, NULL);
```

Miután ez megtörtént, meg kell határozni az eszközöket. Jelen esetben csak egy GPUt akarunk használni.

```
cl_device_id clDevice;
clGetDeviceIDs(clPlatform, CL_DEVICE_TYPE_GPU, 1,
    &clDevice, NULL);
```

Létre kell hozni a context-et.

```
cl_context clGPUContext;
clGPUContext = clCreateContext(0, 1, &clDevice,
    NULL, NULL, NULL);
```

Majd pedig a parancssort szükséges létrehoznunk. Amennyiben több eszközön dolgozunk, úgy ahány eszköz van a rendszerben, annyi parancssort kell létrehozni. Mivel ennél a példánál csak egy GPU-n dolgozunk, így csak egy parancssort kell definiálnunk.

```
cl_command_queue clCommandQueue;
clCommandQueue =
    clCreateCommandQueue(clGPUContext, clDevice,
    0, NULL);
```

Fontos a context-ben létre hozni olyan memória objektumokat, melyek bufferként szolgálva valósítják meg a host, illetve az eszközök közötti kommunikációt.

```
cl_mem hDeviceMemInput, hDeviceMemOutput,
    hDeviceMemMask;
hDeviceMemInput = clCreateBuffer(clGPUContext,
    CL_MEM_READ_ONLY, sizeof(unsigned
    char)*inputarea, NULL, NULL);
hDeviceMemMask = clCreateBuffer(clGPUContext,
    CL_MEM_READ_ONLY,
    sizeof(float)*mask_size*mask_size, NULL,
    NULL);
hDeviceMemOutput = clCreateBuffer(clGPUContext,
    CL_MEM_WRITE_ONLY, sizeof(unsigned
    char)*outputarea, NULL, NULL);
```

A context-en létrehozzuk a kernel programot, illetve azt le kell fordítani, mielőtt azt futtatni kívánjuk. Ezen programban csak egy kernel kód található, de lehetőségünk van, több kernel kódból összeállítani a programunkat.

```
cl_program clProgram;
clProgram =
    clCreateProgramWithSource(clGPUContext, 1,
    &kernel, NULL, NULL);
clBuildProgram(clProgram, 1, &clDevice, NULL,
    NULL, NULL);
```

A kernel kódon belül meg kell határozni, hogy mely függvényt hívja meg program. Esetünkben csak egy függvényről beszélünk, de ettől függetlenül ezt kötelezően meg kell mondanunk.

```
cl_kernel clKernel;
clKernel = clCreateKernel(clProgram,
    "convolution", NULL);
```

A kernel függvénynek, amit az előbb létrehoztunk, fontos a paramétereit meghatározni.

```
clSetKernelArg(clKernel, 0, sizeof(cl_mem),
    (void *)&hDeviceMemInput);
clSetKernelArg(clKernel, 1, sizeof(int),
    (void *)&input_width);
clSetKernelArg(clKernel, 2, sizeof(cl_mem),
    (void *)&hDeviceMemMask);
clSetKernelArg(clKernel, 3, sizeof(int),
    (void *)&mask_size);
clSetKernelArg(clKernel, 4, sizeof(cl_mem),
    (void *)&hDeviceMemOutput);
clSetKernelArg(clKernel, 5, sizeof(int),
    (void *)&output_width);
```

A már létrehozott input buffereket fel kell tölteni a szükséges adatokkal. Ez a művelet már a parancssor feladatkörébe tartozik.

```
clEnqueueWriteBuffer(clCommandQueue,
    hDeviceMemInput, CL_FALSE, 0, sizeof(unsigned
    char)*inputarea, input, 0, NULL, NULL);
clEnqueueWriteBuffer(clCommandQueue,
    hDeviceMemMask, CL_FALSE, 0,
    sizeof(float)*mask_size*mask_size, mask, 0,
    NULL, NULL);
```

Ezután el kell indítani a kernelt. Itt kell megadni, hogy kétdimenziós reprezentációt használunk. A `globalWorkSize` fogja tartalmazni azt, hogy összesen mennyi munkaegység lesz dimenziónként, a `localWorkSize` pedig, hogy egy munkacsoporton belül mennyi munkaegység lesz dimenziónként. Természetesen a `globalWorkSize`-nak egészszámú többszörösének kell lennie a `localWorkSize`-nak, aminek következtében nem mindig pont akkora lesz, mint a kép mérete, amire majd a kernel kódon belül kell majd figyelnünk. Természetesen úgy kell majd kalkulálni a `globalWorkSize`-ot, hogy nagyobb legyen minden dimenzióban, mint a kép.

```
clEnqueueNDRangeKernel(clCommandQueue, clKernel,
    2, 0, globalWorkSize, localWorkSize, 0, NULL,
    NULL);
```

Végül az eredményt visszaolvashatjuk a host-ra. Ennek a lépésnek az elvégzésénél, fontos hogy az összes eredmény ki legyen számítva, emiatt a rendszerbe itt be van építve egy automatikus szinkronizáció, melyet a harmadik paraméter `CL_TRUE`-ra állításával kapcsolhatunk be.

```
clEnqueueReadBuffer(clCommandQueue,
    hDeviceMemOutput, CL_TRUE, 0, sizeof(unsigned
    char)*outputarea, output, 0, NULL, NULL);
```

5.5. Kernel kód

A host kódokhoz mindig tartozik, egy vagy több kernel kód is, mely vezérli az eszközöket.

A kernel kód egy szöveges formátumú konstansként jelenik meg a kódunkban. A host kódban a kernel program létrehozása és fordítása után áll majd elő a GPU számára értelmezhető végrehajtható kód. A kernel sztring C nyelvű eljárások, függvények sorozataként áll elő. Paraméterként kaphat `__global`, `const __global` illetve `__local` jelzéssel ellátott típusokat, előbbi a globális memóriára utal, az azt követő a konstans, végül pedig a

lokális memória jelölését mutatja. Ha ezek egyike se szerepel a paraméter típusa előtt, akkor az a privát memóriát fogja jelenteni.

A következő kódrészletben az előző példa (*lásd: cd melléklet: con_orig.cpp*), azaz a konvolúcióhoz tartozó kernel kód kerül bemutatásra. Ez a kód a DRSCREEN rendszer számos pontján (például érhálózat detektálása) felhasználható.

```
__kernel void convolution (
    const __global unsigned char* input,
    int input_width,
    const __global float* mask,
    int mask_size,
    __global unsigned char* output,
    int output_width)
{
const int gid_x = get_global_id(0);
const int gid_y = get_global_id(1);
int i,j;
float sum = 0.0;
for (i=0; i<mask_size; i++) {
    for (j=0; j<mask_size; j++) {
         sum +=
              input[(gid_y+i)*input_width+gid_x+j] *
              mask[i*mask_size+j];
    }
}
output[gid_y*output_width+gid_x] = sum;
}
```

A `get_global_id(i)`-vel az adott munkaegységnek i. (i = 0...k-1) dimenzióbeli pozícióját tudhatjuk meg. Ennek segítségével határozhatjuk meg a programom belül betöltött globális pozícióját az adott munkaegységnek. Az egyes tömbök indexelésénél is gyakran használják ezt az eszközt.

A kódban a `convolution` eljárás miután meghatározta a globális pozícióját, a megfelelő méretű környezetében lévő képrészleten végighaladva, az ott található értéket megszorozva a maszkon megfelelő értékével, hozzáadja egy `sum` nevű változóhoz, mely a ciklusok végén tartalmazza a megfelelő eredményt, melyet ezután az `output` megfelelő helyére be kell írni.

6. A Kernel kód létrehozásának lehetőségei

A programunk kernel kódjának nem feltétlenül kell konstansnak lennie. Különböző módszerek vannak, amivel befolyásolhatjuk akár működését, akár magát a kódot. Mivel a kernel kódunk futási időben lesz lefordítva, így lehetőségünk van arra, hogy a fordítás pillanatáig manipulálhassuk.

6.1 Statikus és dinamikus kernel kódok

Statikus kernel kódról akkor beszélhetünk, ha az eredi programban nincs semmilyen a kernel kódot, vagy annak valamely változóját, konstansát változtató kódrészlet, ellenkező esetben dinamikusnak tekinthető a kernel.

Többféleképpen is tudjuk befolyásolni a kernel kódot. Az egyik lehetőség, hogy a fordítás közben adunk meg olyan konstansokat, melyek a program futása közben határozódtak meg (*lásd: cd melléklet: con_cons.cpp*). A fentebbi konvolúciós példában ilyen lehet az `input_width`, `mask_size`, és az `output_width`. Az eredeti kódunkban ezek paraméterként kerülnek meghatározásra, viszont nyilvánvalóan egy adott kép és maszk esetén ezek a számok konstansak.

Minden egyes szálon teljesen fölösleges és erőforrás pazarló ezen értékeket átadni, így célszerű ezeket konstansként megadni. A `clBuildProgram` negyedik paraméterénél, ami `options` névre hallgat, meg lehet adni különböző, a fordítónak szóló kapcsolókat, melyek közül ebben az esetben a `-D name = definition` érdekes számunkra. A `name` a konstansunk nevét jelöli, a `definition` pedig az értékét.

Ha az alábbira módosítjuk a kernel kódunkat, akkor a fentebb említett paramétereket átalakítjuk úgymond még definiálatlan konstansokká:

```
__kernel void convolution (
    const __global cl_uchar* input,
    const __global cl_float* mask,
    __global cl_uchar* output)
{
const gid_x = get_global_id(0);
const gid_y = get_global_id(1);
cl_int i,j;
cl_float sum = 0;
for (i=0; i<MASK_SIZE; i++) {
    for (j=0; j<MASK_SIZE; j++) {
         sum +=
              input[(gid_x+i)*INPUT_WIDTH+gid_y+j] *
              mask[i*MASK_SIZE+j];
    }
}
output[gid_x*OUTPUT_WIDTH+gid_y] = sum;
}
```

A konstansaink fordítás közben definiáltakká válnak, ha a `clBuildProgram` `options` paraméterének a következőt adjuk:

```
"-DINPUT_WIDTH=500 -DMASK_SIZE=21
    -DOUTPUT_WIDTH=480"
```

Ezután a fordító a kódunkba a konstansok helyére beírja az adott értéket és azokkal fordítja le. Ezzel a technikával további gyorsulásokat lehet elérni, habár ebben a példában ez nem jelentős mértékű.

6.2 Generált dinamikus kernel kód

Egy másik lehetséges módja a dinamikus kernel kódok létrehozásának, hogy a program futása során generáljuk a kernel kódunkat (*lásd: cd melléklet: con_gene.cpp*). Ekkor a program futása kezdetén nincsen még meg a teljes kernel kódunk, annak bizonyos sorai hiányoznak. Ezek a sorok a program futása közben kerülnek bele, viszont a fordításig teljessé kell válniuk.

A konvolúciós példánál maradva, például a sum előállításánál minden esetben a `mask` tömb ugyanazokat az értékeket tartalmazza. Ezen felül az

`input` kép pontjainak egymáshoz viszonyított helyzete is konstans, csak a kép szélességétől függ.

A `mask` tömböt akár konstansként is kezelhetnénk, viszont a fentebbi technikával nem tudnánk megvalósítani, mivel az elemszáma változó lehet, és ugyanez igaz az input tömb indexelésével kapcsolatban is.

Elő tudunk viszont olyan sorokat állítani, melyek tartalmazzák mind a kiszámított indexeket, mind a megfelelő maszk értékeket.

```
for (i=0; i<mask_size; i++) {
    for (j=0; j<mask_size; j++) {
        sprintf (buffer,
            "sum += input[gid+%d] * %.10f;\n",
            i*in_w+j, mask[i*mask_size+j]);
        ker += buffer;
    }
}
```

Ekkor a fenti iteráció minden lépésénél a `kernel` sztringet a kódrészlet kibővíti a `buffer` sztring sorával, mely tartalmazza a relatív indexelését az `input` tömbnek, mint konstansként a `mask` adott értékét. Ezzel a technikával generálhatjuk a kernel kódunk sorait, melynek eredménye az alábbi képen fog bekerülni a kernel kódba, persze a paraméterektől függően és iterációnként más-más számokkal.

```
sum += input[gid+0] * 0.1111111119;
sum += input[gid+1] * 0.1111111119;
...
```

Ennek a technikának több előnye is lehet, melyek a program gyorsulását is eredményezhetik. Például ebben az esetben, lényeges gyorsulást eredményezett az, hogy nem kellett minden iterációs lépésnél kiszámolni az `input` tömb indexelést. Igaz, hogy csak egy összeadást és egy szorzást spóroltunk, de egy pixel kiszámításához egy 21×21es maszk esetén ez már több mint 880 művelet. Ezen felül megspóroltuk mind a `mask` index kiszámítását, mind a `mask` tömbből való beolvasását is.

Maga a kernel kód generálása is időbe telik, de ez csak körülbelül 1-10 milliszekundum a kód hosszától függően, de ehhez képest lényegesen, nagyságrendekkel gyorsabb kódot kapunk.

6.3 Bináris kernel kód

A párhuzamos kódok létrehozásánál, az is lényeges kérdés, hogy a párhuzamos kód miatt keletkező adminisztráció mekkora erőforrást igényel. Ha túl sok ideig tart ahhoz képest, hogy mennyit tudunk javítani a párhuzamos kódok használatával, akkor nem éri meg átalakítani a kódunkat.

Ebből a gondolatmenetből kiindulva lemértem, hogy az egyes adminisztrációs lépések, mennyi időt vesznek idénybe. Ez alapján készítettem az alábbi táblázatot, melyben az átlagos idő található milliszekundumban.

Platform meghatározása	<1 ms
Eszköz meghatározása	<1 ms
Context létrehozása	40 ms
Parancssor létrehozása	<1 ms
Program létrehozása	<1 ms
Program fordítása	250 ms
Kerner kód megadása	<1 ms
Argumentumok beállítása	<1 ms
Buffer írás	<1 ms
Kernel elindítása	<1 ms
Buffer olvasása	erősen kódfüggő

A platform meghatározásnál az az érdekes eset fordult elő rendszeresen, hogy az első lefutás alkalmánál mindig cirka 70 milliszekundum ideig tartott, viszont ezt leszámítva 1 milliszekundum alatt végrehajtódott. Ez amiatt következik be, hogy az első lefutáskor még nem történt meg az operációs rendszer szükséges állományainak beolvasása.

A buffer eredményének visszaolvasása a kernel kód futási idejétől függ, mivel itt van egy automatikus szinkronizáció, ami addig nem engedi végrehajtani ezt a műveletet, amíg az összes szál be nem fejezte működését. Emiatt a tény miatt, ennek az ideje nem az adminisztrációhoz tartozik, hanem valójában a kernel kód futási idejének a méréséhez tartozik.

A legtöbb adminisztráció, a fentieket leszámítva, gyorsan végrehajtódik, kivétel ez alól a context létrehozása, illetve a program fordítása. Előbbivel nem nagyon lehet mit kezdeni, utóbbinál viszont lehet olyan megoldást találni, melynél ez az idő lecsökkenthető.

Az eddigi programoknál a kernel kód egy szöveges C kód volt, melyből a kernel programot a `clCreateProgramWithSource` metódussal hoztunk létre. Ezen kívül lehetőségünk van arra, hogy a kernel programot a `clCreateProgramWithBinary` metódussal hozzuk létre. Ennél a módszernél viszont, nem egy C nyelvű kódot vár a program, hanem egy előfordított assembler kódot.

Az assembler kód speciális GPU utasításokat tartalmaz. Ezen tulajdonsága miatt, amennyiben ilyen kódokat akarunk írni, úgy az OpenCL heterogenitásáról le kell mondanunk. Valamint tapasztalataim alapján, nem csak a más gyártó által készített video kártyákon, és CPUkon, hanem ugyanazon a hardveren sem működik megfelelően, ha más verziójú driver van feltelepítve az adott gépre. Így ez a módszer nem alkalmazható akkor, ha fontos szempont, hogy heterogén környezetben is fusson a kódunk, illetve ha szállítható kódot kell írnunk. Ezen problémák áthidalására nem sikerült még megfelelő megoldást találnom.

Ha viszont nem fontosak a fent leírt szempontok, akkor alkalmazható ez a megoldás is. A hátrányokon kívül előnyei is vannak ennek a módszernek, melyek közül a leglényegesebb az, hogy az assembler kódból létrehozott program fordítási ideje 5 milliszekundum alatt van. Így akár több mint 245 milliszekundummal is csökkenthetjük programunk futási idejét, ami programtól függően akár nagyságrendi gyorsulást is jelenthet.

Az assembler kód egyedisége, illetve alacsony szintje miatt a programozhatósága is jelentősen romlik, viszont erre sikerült egy viszonylag egyszerű megoldást találni. Amennyiben készítünk egy hagyományos C alapú kernel kódot, és azt lefordítjuk, akkor az alábbi eljárással kinyerhetjük az adott program binárisát, azaz az assembler kódját.

```
clGetProgramInfo(cpProgram, CL_PROGRAM_BINARIES,
    size, bin, NULL);
```

A `bin` egy sztring tömb, melynek annyi eleme kell, hogy legyen, ahány eszközt definiáltunk a program futása során. A mi esetünkben jelenleg csak egy GPU áll rendelkezésre, így a `bin[0]` helyén megtalálhatjuk azt a sztringet, mely tartalmazza a bináris, assembler kódot. Nyilvánvalóan egy assembler kódot nehezebb módosítani, mint egy C kódot, így ezt a módszert akkor célszerű csak alkalmazni, ha biztosak vagyunk abban, hogy jól működő, szemantikailag helyes kódunk van.

Habár nehezebben megvalósítható és nagyobb odafigyelést igényel, de a bináris kernel kódot is elő lehet állítani generálási módszerrel. Ez a technika, vagyis a generált bináris kód alkalmazása, kísérleti jelleggel implementálásra is került a bikubikus interpoláció alkalmazásba (*lásd: cd melléklet: bicubic.cpp*). Ennek az algoritmusnak a segítségével a képeket fel lehet nagyítani és a kapott eredmény jól közelíti az optikai nagyítás eredményét. Ezt a módszert a DRSCREEN projekt számos algoritmusánál használják. Egy 500×500 kép 2,4-szeres nagyítása esetén, a MATLAB 457 milliszekundum alatt futott le, míg a saját 73 milliszekundum alatt, ami több mint hatszoros gyorsulás eredményezett. Amennyiben nem bináris kódot használtunk volna, akkor 245 milliszekundummal tovább futna a program, azaz összesen 318 milliszekundumig futott volna, ami már nem olyan kiugró eredmény. Természetesen, ha csak a kernel futási idejét nézzük, azaz nem számoljuk az adminisztrációt, a képkibővítését, és a szemétgyűjtögetést, akkor az átlagos futási idő 16 milliszekundum, ami 28-szoros sebességnövekedést jelent.

7. Összegzés

Munkám eredményeként egyes esetekben valódi gyorsulást sikerült elérni. A korábban említett konvolúciónál, egy jobban optimalizált kódot tekintve egy 500×500-as képen 21×21-es maszk esetén a GPU-n átlagosan 0.01 másodperc alatt futott le a kernelkód, míg a hagyományos módszerrel megírt CPU-n futó kód átlagosan 0.55 másodpercet igényelt a végrehajtáshoz. Ez megközelítőleg 55-szörös gyorsulás, ami már jelentősnek mondható. Ha az algoritmusok többségénél nem is ilyen jelentős, de mondjuk átlagosan 10-szeres gyorsulást sikerülne elérni, akkor az az egész projekt szempontjából jelentős előrelépés lenne.

Az átállás, azaz már egy meglévő környezetbe való beépítés sajnos komoly problémákat okozott, de szerencsére sikerült jó megoldást találni a kezelésére. Sikerült azt is elkerülni, hogy ez a jelentős technikai előrelépés kimaradjon, és azt is, hogy az egész projektet teljesen elölről kelljen kezdeni. A MEX és a benne rejlő lehetőségek, remek megoldást nyújtottak, viszont még vannak megválaszolatlan kérdések ezzel a technikával kapcsolatban, főleg azon a téren, hogy ezen belül van-e lehetőség további optimalizációkra. A legfontosabb azonban az a tény, hogy működőképes a rendszer.

7.1 Az algoritmusok vizsgálata

Ez a lépés, mint korábban is kifejtettem eléggé fontos, főleg, hogy ezáltal kiszűrhetjük az olyan kódokat melyeknél vagy nincs is gyorsulás vagy elhanyagolható méretű, és így a GPUra történő átírás fölöslegessé válhatna.

Amiatt is fontos ez a vizsgálat, hogy egy becslést tudjunk készíteni, arról hogy az algoritmusok közül melyeknél érhetünk el látványos javulást. A rövidebb ideig futó programok esetén különösen oda kell figyelni, ugyanis az adminisztrációs idő akár túlnőhet a technológia által elért eredményeken.

Ezenfelül ha még figyelembe vesszük az algoritmusok gyakoriságát is, azaz, hogy melyiket milyen gyakran használja az egész projekt, úgy a globális, az egész projektre kiterjedő gyorsulás már az átállás elején jelentős lehet.

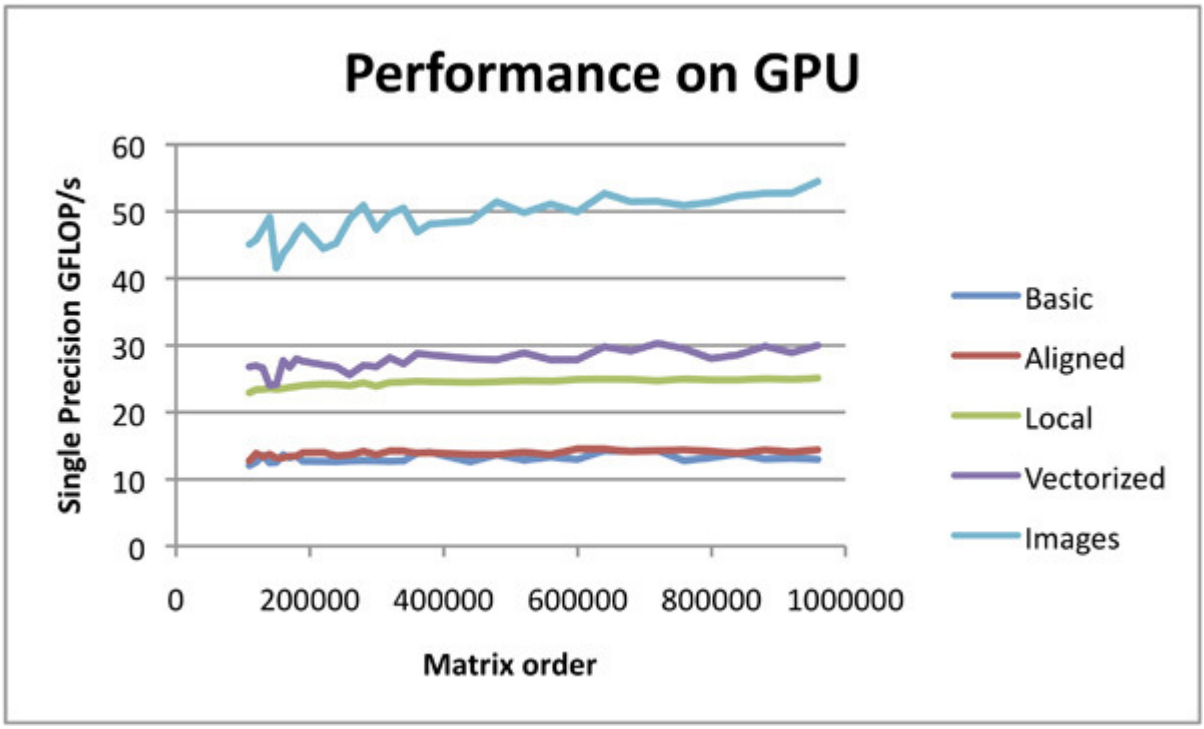
Eddigi munkálataim során főleg ezzel kapcsolatos vizsgálatok elvégzése, a projektben szereplő algoritmusok (lásd Függelék) párhuzamosíthatóságának vizsgálata, illetve azon belül a GPU-n történő megvalósíthatóság lehetőségének feltérképezése volt. Ezen kívül egy olyan megoldás kellet keresnem, amelynek segítségével ezt a technológiát be lehet építeni a jelenlegi projektbe. A projekten belül a jövőben ezek alapján történő implementálások elvégzése a fő szempont.

7.2. További gyorsítási lehetőségek

A memóriák közti különbségekből adódóan további jelentős gyorsulás érhető el, ha lokális vagy privát memóriákat lehetőség szerint minél többször használunk. Ennek megfelelően programjaink írásakor ez egy másik fontos szempont, mely a jövőbeli kutatásoknál, programírásoknál kiemelt szerepet kell, hogy majd kapjon. Jelenleg ennek a fontossága az algoritmusok vizsgálatánál, még nem jelent meg. A probléma áttekintése nagyobb körültekintést igényel, mivel nem minden adat kerülhet át a lokális memóriába. A korábbi konvolúciós példámat tekintve például az input kép nem kerülhetne át, mivel nem lehet egy konkrét határvonalat húzni ameddig csak az egyik munkacsoport dolgozik, és onnantól pedig egy másik. Viszont lehetőség van arra, hogy csak egy adott, a munkacsoportnak szükséges terület, másolódjon át, noha ezzel megnő a szükséges memória terület. Ha sikerül ezekre a kérdésekre választ találni, akkor egy újabb támpont nyílhat a vizsgálatok során.

Emellett a legújabb kutatások alapján ennél jelentősebb gyorsulást is el lehet érni, azáltal hogy adatainkat szó szerint képként kezeljük. A GPU-k architektúrája egy-két ponton eltér az OpenCL logikai felépítésétől. Ezt

kihasználva az AMD kutatói olyan megoldást [15] készítettek, mely az adatokat képként közvetlenül a GPU textúra memóriájában tárolja, melynek elérése a lokális memória elérési idejével közel megegyező, viszont a másolás költségét megkerülve jobb teljesítményt értek el (*6. ábra*). Ők viszont arra ők is felhívták a figyelmet, hogy ez nem minden eszközön működik, de ha csak egy céleszközön akarjuk futtatni a kódjainkat, akkor ez a megoldás is alkalmazásra kerülhet a későbbiekben.



6. ábra: A textúra memória használatával elért gyorsulás eredmény (forrás: http://developer.amd.com/documentation/articles/Pages/OpenCL-Optimization-Case-Study_8.aspx, utolsó megtekintés: 2011. április)

Köszönetnyilvánítás

Mindenképp szeretném megköszönni konzulensemnek, Dr. Zichar Marianna tanárnőnek, hogy a félévek folyamán útmutatást adott nekem, valamint a kérdéseimre válaszolva elősegítette a diplomamunkám elkészítését.

A dolgozatban közölt eredmények eléréséhez szükséges kutatást az NKTH TECH08-2 „DRSCREEN – A cukorbetegség szemszövődményeinek szűrésére alkalmas képfeldolgozó rendszer kifejlesztése” című, OM-00194/2008, OM-00195/2008, OM-00196/2008 szerződésszámú projekt támogatta.

Irodalomjegyzék

[1] NVIDIA® Corporation, (Santa Clara, CA, USA). http://www.nvidia.com/content/global/global.php (utolsó megtekintés: 2011. április)

[2] A Tesla™ C2070 Computing Processor az NVIDIA® cég terméke. http://www.nvidia.com/object/product_tesla_C2050_C2070_us.html (utolsó megtekintés: 2011. április)

[3] NKTH TECH08-2 „DRSCREEN – A cukorbetegség szemszövődményeinek szűrésére alkalmas képfeldolgozó rendszer kifejlesztése" című, OM-00194/2008, OM-00195/2008, OM-00196/2008 szerződésszámú projekt

[4] A CUDA™ az NVIDIA® cég terméke. http://www.nvidia.com/object/cuda_home_new.html (utolsó megtekintés: 2011. április)

[5] Khronos Group, (Beaverton, Oregon, USA). http://www.khronos.org/ (utolsó megtekintés: 2011. április)

[6] OpenCL™ a Khronos Group API-ja. http://www.khronos.org/opencl/ (utolsó megtekintés: 2011. április)

[7] CMSoft: Overview about OpenCL and parallel processing. http://www.cmsoft.com.br/index.php?option=com_content&view=category&layout=blog&id=59&Itemid=106 (utolsó megtekintés: 2011. április)

[8] AMD®, Advanced Micro Devices Inc. (Sunnyvale, CA, USA). www.amd.com (utolsó megtekintés: 2011. április)

[9] A MATLAB® szoftver a MathWorks (Natick, MA USA) cég terméke, www.mathworks.com (utolsó megtekintés: 2011. április)

[10] AccelerEyes (Atlanta, GA, USA) http://www.accelereyes.com/ (utolsó megtekintés: 2011. április)

[11] A Jacket® szoftver az AccelerEyes cég terméke. http://www.accelereyes.com/products (utolsó megtekintés: 2011. április)

[12] MEX, a MATLAB® speciális fordítója. http://www.mathworks.com/support/tech-notes/1600/1605.html (utolsó megtekintés: 2011. április)

[13] Az OpenCL™ 1.1-es verziójának hivatalos referenciája Szerkesztő: Aaftam Munshi http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf (utolsó megtekintés: 2011. április)

[14] Sofka M, Stewart CV. Retinal vessel extraction using multiscale matched filters confidence and edge measures. IEEE Transactions on Medical Imaging. 2005;25 No. 12,

[15] Bryan Catanzaro: OpenCL™ Optimization Case Study: Diagonal Sparse Matrix Vector Multiplication http://developer.amd.com/documentation/articles/Pages/OpenCL-Optimization-Case-Study.aspx (utolsó megtekintés: 2011. április)

CD melléklet tartalma

- `GPUconv.cpp` – A konvolúciós példaprogram (`con_gene.cpp`) felkészítve MEX-xel történő fordításhoz. (4.2 fejezet)
- `GPUconv.mexw32` – A konvolúciós példaprogram bináris m fájlként előállt verziója (`GPUconv.cpp`-ből fordított), melyet MATLAB-ban meg tudunk hívni. (4.2 fejezet)
- `con_orig.cpp` – A konvolúciós példaprogram első, eredeti forráskódja. (5. fejezet)
- `con_orig.exe` – A `con_orig.cpp` lefordításából előlát futtatható állomány.
- `con_cons.cpp` – A konvolúciós példaprogram forráskódja, definiálatlan konstansok használatával. (6.1 fejezet)
- `con_cons.exe` – A `con_cons.cpp` lefordításából előlát futtatható állomány.
- `con_gene.cpp` – A konvolúciós példaprogram forráskódja, generált kernelkód alkalmazásával. (6.2 fejezet)
- `con_gene.exe` – A `con_gene.cpp` lefordításából előlát futtatható állomány.
- `bicubic.cpp` – A bikubikus interpoláció forráskódja. (6.3 fejezet)
- `bicubic.exe` – A `bicubic.cpp` lefordításából előlát futtatható állomány.
- `notes.txt` – A fentebbi C++ kódok fordításához szükséges, és egyéb hasznos információkat tartalmazz.

Függelék: A DRSCREEN rendszer algoritmusai

Az alábbi táblázat látható a projektben szereplő algoritmusok párhuzamosíthatósági szintjei és GPU-n való megvalósíthatósága.

Algoritmus neve	Párhuzamosíthatósági szintek	GPU-n implenetálható részek
Fundus image preprocessing		
Contrast enhancement		
- PRE_WK	Pixel szintű	Gauss szűrés
Shade correction		
- PRE_SHADE	Pixel szintű	Medián szűrés
Fundus image prefiltering		
Lesion candidate detection		
- PRF_CAN	Pixel szintű	Medián szűrés
Left/Right eye detection		
- PRF_LRD	Pixel szintű	Küszöbölés
Orientation classification		
- PRF_ORI 6	Pixel szintű	Sajátságkinyerés
Quality assessment		
- PRF_QAB	Pixel szintű	Sajátságkinyerés
- PRF_QAH	Pixel szintű	Sajátságkinyerés
Severity filtering		
- PRF_SEV	Pixel szintű	Sajátságkinyerés
Anatomical parts detection		
ROI detection		
- ROI_GAGNON	Pixel szintű	VAGY operátor
- ROI_HMF	Pixel szintű	Gradiens számolás
Vascular system detection		

- VS_KNNPC	Pixel szintű	Küszöbölés
- VS_M	Pixel szintű	Medián szűrés, Morfológia, Küszöbölés
- VS_M2	Pixel szintű	Medián szűrés, Morfológia, Küszöbölés
- VS_HMF	Pixel szintű	Gradiens számolás
Optic disc detection		
- ODC_DECOMPOSITION	Pixel szintű	Konvolúciós maszkolás
- ODC_EDGE	Pixel szintű	Konvolúciós maszkolás
- ODC_ENTROPYFILTER	Pixel szintű	Konvolúciós maszkolás
- ODC_FUZZY	Pixel szintű	Vékonyítás
- ODC_HOUGH	Pixel szintű	Morfológia; Medián szűrés;
Macula detection		
- MA_REGION	Pixel szintű	Küszöbölés
- MA_SHADE	Pixel szintű	Medián szűrás
- MA_WATERSHED	Részképi szintű	Morfológia
Optic disk, Macula combined system		
- ODMA_CSYS	Részképi szintű	Távolság mérés
Lesions detection		
Microaneurysm detection		
- MD_ABRAMOFF	Részképi szintű	Sajátságkinyerés osztályozás
- MD_FLEMING	Részképi szintű	Sajátságkinyerés osztályozás
- MD_WALTER	Képi szintű	-
- MD_NCUT	Részképi szintű	Sajátságkinyerés osztályozás

- MD_ZHANG	Részképi szintű	Sajátságkinyerés osztályozás
- MD_LAZAR	Részképi szintű	Sajátságkinyerés osztályozás
- MD_HOUGH	Részképi szintű	Sajátságkinyerés osztályozás

Plágium - Nyilatkozat

Szakdolgozat készítésére vonatkozó szabályok betartásáról nyilatkozat

Alulírott Farkas László (Neptunkód: J61X9I) jelen nyilatkozat aláírásával kijelentem, hogy a

Komplex képfeldolgozó rendszer algoritmusainak GPU támogatása

című diplomamunka

(a továbbiakban: dolgozat) önálló munkám, a dolgozat készítése során betartottam a szerzői jogról szóló 1999. évi LXXVI. tv. szabályait, valamint az egyetem által előírt, a dolgozat készítésére vonatkozó szabályokat, különösen a hivatkozások és idézések tekintetében.

Kijelentem továbbá, hogy a dolgozat készítése során az önálló munka kitétel tekintetében a konzulenst, illetve a feladatot kiadó oktatót nem tévesztettem meg.

Jelen nyilatkozat aláírásával tudomásul veszem, hogy amennyiben bizonyítható, hogy a dolgozatot nem magam készítettem vagy a dolgozattal kapcsolatban szerzői jogsértés ténye merül fel, a Debreceni Egyetem megtagadja a dolgozat befogadását és ellenem fegyelmi eljárást indíthat.

A dolgozat befogadásának megtagadása és a fegyelmi eljárás indítása nem érinti a szerzői jogsértés miatti egyéb (polgári jogi, szabálysértési jogi, büntetőjogi) jogkövetkezményeket.

hallgató

Debrecen, 2011. április 18.