Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Távközlési és Médiainformatikai Tanszék

ROBUSZTUS GPGPU PLUGIN FEJLESZTÉSE A *RAPIDMINER* ADATBÁNYÁSZATI SZOFTVERHEZ

KÉSZÍTETTE: KOVÁCS ANDOR

EMAIL CÍM: VERMILION.ANDOR@GMAIL.COM

KONZULENS: PREKOPCSÁK ZOLTÁN (TMIT)

EMAIL CÍM: PREKOPCSAK@TMIT.BME.HU

KIVONAT

Az elmúlt években jelentős számú cikk jelent meg olyan grafikuskártya programokról, melyek CPU-s megfelelőjükhöz képest, akár több nagyságrendű gyorsulást is fel tudtak mutatni[*1*][*2*][*3*]. Ezen cikkek olvasása közben három fontos kérdés merül fel a téma iránt érdeklődő olvasóban:

-A jelentős gyorsulások ellenére ténylegesen mennyire optimálisak ezek a szoftverek?

-A gyorsulások mennyire nevezhetők robusztusnak, fenntarthatónak, ha a bemenet mérete nagyságrendekkel nő?

-A jelenlegi fejlesztői eszközökkel lehetőség van-e ezek beépítésére komoly ipari szoftverekbe, úgy hogy CPU-hoz viszonyított gyorsaságukból még akkor se veszítsenek, ha a feldolgozandó adatmennyiséget több nagyságrenddel növeljük?

Dolgozatom ezekre kérdésekre próbál minél részletesebb és pontosabb választ adni. Ehhez kiválasztottam a k legközelebbi szomszéd algoritmust, amihez már számos GPGPU implementáció készült, illetve napjaink egyik legnépszerűbb adatbányászati eszközét a RapidMiner-t, hogy ezeken keresztül vizsgáljam meg egy GPU plugin fejlesztésének lehetőségeit, illetve limitációit.

Munkám során célom volt olyan megoldások kidolgozása, melyekkel kiküszöbölhetem azon hiányosságokat és korlátokat, amikkel a feljebb említett cikkek írói nem foglalkoztak, pedig ezen korlátok miatt a szoftvereik széleskörű ipari alkalmazásra korlátozottan vagy teljesen alkalmatlanná válnak.

A munkám eredménye egy olyan plugin, ami sebességben felveszi a versenyt a jelenlegi leggyorsabb megoldásokkal, azonban robusztusságban jelentősen túlmutat azokon, akár gigabájtos nagyságrendű fájlok feldolgozására is alkalmas.

Új lehetőségeket ad az adatbányászoknak, akik mindezt könnyedén, a RapidMiner grafikus felületén keresztül érhetik el, és GPU specifikus tudás nélkül használhatják ezt a CPU implementációhoz hasonló módon.

Fejlesztéshez az Nvidia Cuda nyelvet és annak Java illesztését, a JCuda-t[*4*] használtam.

TARTALOM

1. **Bevezetés** 4

2. **A CUDA programok hardveren történő futásának részletes ismertetése** 6

2.1 Az architektúra alapelvének bemutatása 6

2.2 Compute Capability, kártyák különbségei 7

2.3 Streaming Multiprocesszorok 8

2.4 További cuda optimalizációs eszközök 17

2.5 A CUDA alkalmazások fordításának lehetőségei: 18

3. **Létező CUDA k-NN megoldások, publikációk** 19

3.1 K legközelebbi szomszéd algoritmus 19

3.2.: Létező Megvalósítások 20

4. **A plugin tervezése:** 25

4.1 A RapidMiner rövid bemutatása 25

4.2 A RapidMiner k-NN funkciójának elemzése 26

4.3 A RapidMiner és a CUDA összekötése, kommunikációja 28

5. **JCUDA implementáció elkészítése** 30

5.1 Adathalmazok tárolás a GPU-n 31

5.2 Távolságmátrix számítása 33

5.3 Legkisebb k érték keresése 35

5.4 A megfelelő címke kiválasztása, Általános működés 37

5.5 A probléma feldarabolása, aszinkron működés: 37

6. **Mérések** 39

6.1 Távolságmátrix számítás Kísérleti számítások 39

6.2 Összehasonlítás a korábbi megoldással 42

6.3 végleges implementációk tesztje 44

6.4 Legkisebb K elem kiválasztására adott módszer mérése 46

6.5 Kiegészítő funkciókkal kapcsolatos mérések 47

6.6 összehasonlítás a CPU-s megfelelővel 50

6.7 A kész plugin technikák között választó döntési fái: 51

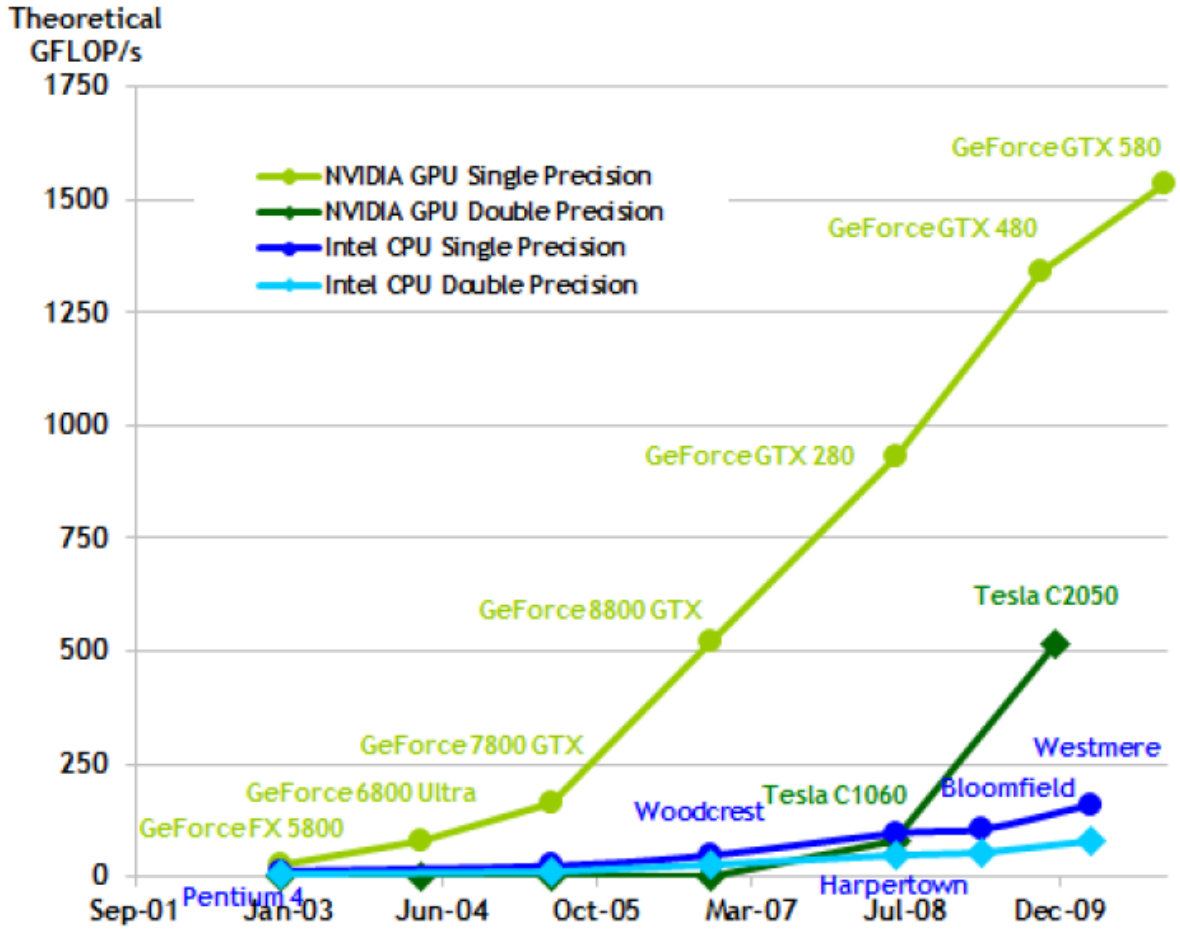
7. **Összefoglalás, jövőbeli tervek** 53

IRODALOMJEGYZÉK 54

1. BEVEZETÉS

Az elmúlt években jelentős számú cikket olvashattunk, általános célú grafikuskártya programozás (GPGPU) témakörben[*1*][*2*][*3*]. Ezekben általában egy vagy több algoritmust implementáltak a GPU-n, majd azt összehasonlították a CPU-s megfelelőjükkel, és akár több nagyságrendű gyorsulást is fel tudtak mutatni. Köszönhető ez annak, hogy ezen kártyák számítási kapacitása az elmúlt években jelentősen túlszárnyalta a processzorokét, ezen különbség megfigyelhető az ***1. ábrán***.

A cikkek olvasása közben, az olvasó valószínűleg eljátszik azon gondolattal, hogy mekkorát tudna abból profitálni, ha ezek a grafikuskártya programok, nem csak tudományos cikkekben szerepelnének, hanem komoly ipari használatra alkalmas szoftverek aktív részeként is elérhetőek lennének. Így akár a GPU programozásban komoly szakértelemmel nem rendelkező személyeknek is előnyére válhatna a bizonyos fent említett több nagyságrendű gyorsulás, illetve az ***1. ábrán*** is láthatóan egyre nagyobb CPU-GPU számítási kapacitás különbség.



1. ÁBRA GPU-CPU LEBEGŐPONTOS SZÁMÍTÁSI KAPACITÁSÁNAK ÖSSZEHASONLÍTÁSA[4]

Ahogy a GPGPU programok futtatására alkalmas grafikuskártyák egyre olcsóbbak lesznek, egyre inkább nő az igény az olyan szoftverekre, amik ki is tudják használni az ezekben rejlő lehetőségeket. Azonban ezen programok, bizonyos alkalmazási területeken meglehetősen sokat váratnak magukra.

Ilyen terület az adatbányászat. Dolgozatomban egy ehhez a területhez tartozó algoritmust, a k legközelebbi szomszédot (k-NN) választottam ki, ami egy széles körben alkalmazott osztályozási, gépi tanulási és minta felismerési módszer, de magas számításigénye komoly korlátokat jelent.

Érdekes, hogy az első CUDA nyelven íródott k-NN implementációt prezentáló cikk már három éve megjelent[*1*], de még napjainkban sincs olyan adatbányászati szoftver, ami képes lenne ezen algoritmust a grafikus kártya segítségével gyorsítani. Felmerül tehát a kérdés, hogy a cikkben szereplő két nagyságrendű gyorsulás mennyire volt valós, és az implementáció mennyire volt robusztus, illetve rugalmas? Kérdezve ezalatt azt, hogy mik lehetnek azok a limitációk, amik miatt, még napjainkban is inkább az akár 200-szor lassabb CPU implementációkat alkalmazzuk?

Dolgozatom fő témája tehát ezen korlátok megtalálása és kiküszöbölése, úgy, hogy közben a bemenet méretétől függetlenül, akár többféle architektúrájú grafikuskártyán is, a lehető legnagyobb gyorsulást érjem el a CPU-s programokhoz képest.

Munkám eredménye, nem csupán egy implementáció, hanem egy könnyen telepíthető plugin is, ami napjaink egyik legnépszerűbb nyílt forráskódú adatbányászati szoftverébe a RapidMinerbe épül be. Lehetővé teszi a felhasználók számára, hogy a lehető legegyszerűbben, GPU specifikus tudás nélkül használják az akár több nagyságrendű gyorsulást a CPU-n futó verzióhoz képest.

A bővítményem grafikuskártyán futó részét Nvidia CUDA nyelven valósítottam meg, azonban, mivel a RapidMiner Java nyelven íródott, az integráláshoz szükségem volt a CUDA nyelv Java illesztésére, a JCUDA-ra.[*5*]

Hogy könnyen rá tudjak világítani az eddigi k-NN implementációk hiányosságaira nagyon részletesen be kell mutatnom a CUDA architektúra működését, ennek következtében azonban a dolgozat terjedelme nem engedi meg a CUDA nyelv ismertetését, az ezt nem ismerők számára ajánlom az önálló laboratórium beszámolómat tömör összefoglalóként.[*19*]

A következő fejezetben, ezen tudásra építve bemutatom, hogy a nyelv miként képződik le a hardverre, az milyen elemekből áll. Erre nagy szükség lesz, hogy a 3. fejezetben bemutathassam, hogy a korábbi implementációk nagy valószínűséggel, napjainkban miért nem szerepelnek komoly szoftverekben, és hogy a 4. fejezetben ezt a tudást felhasználhassam a plugin tervezéséhez.

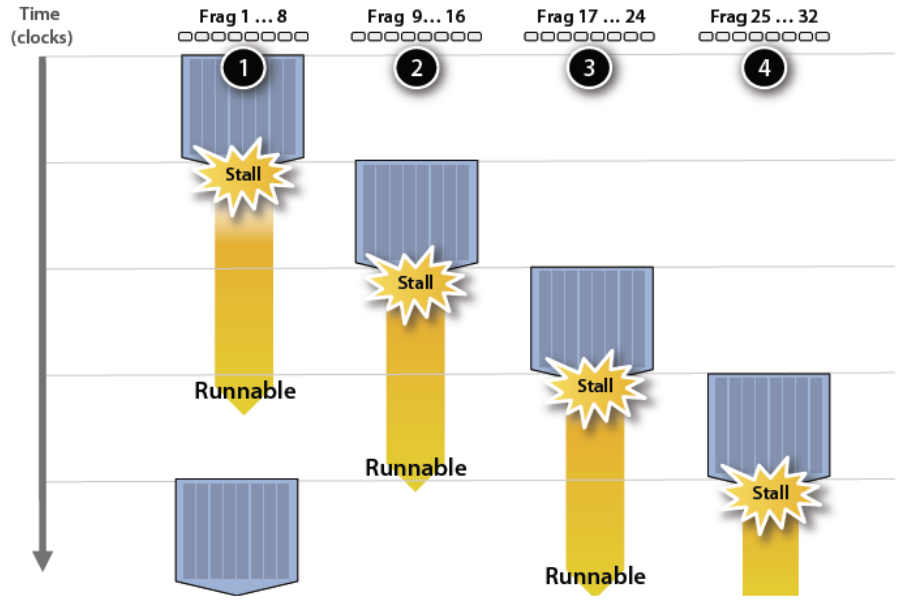
Az 5. fejezetben ezek után ismertetem, hogy milyen módszerekkel hidaltam át a korábbi implementációk gyengeségeit, hiányosságait, ezek hatékonyságát pedig a 6. fejezetben mérésekkel igazolom.

Dolgozatom végén összefoglalom az elért eredményeket, és jövőbeli tervemről nyilatkozom.

2. A CUDA PROGRAMOK HARDVEREN TÖRTÉNŐ FUTÁSÁNAK RÉSZLETES ISMERTETÉSE

2.1 AZ ARCHITEKTÚRA ALAPELVÉNEK BEMUTATÁSA

A CUDA programok futtatására alkalmas GPU-k architektúrája Single Instruction Multiple Thread elven működik.[*6*] Ennek megértéséhez először a futtatás alapelemeit kell megismernünk. A legfőbb elemei az úgynevezett „warp"-ok, ezek 32 szálból álló csoportok, melyek az ütemezés és végrehajtás szempontjából egységet képeznek. A warpokon belül a 32 szál ugyanazt az utasítást hajtja végre, általában más adatokon. Kivételt képeznek ez alól az elágazások a programban, például az *if-else* szerkezetek, melyek futtatása esetén, a warpon belül úgynevezett divergencia lép fel, tehát a szálak egy részére az if ág, a fennmaradó részükre pedig az else ág teljesül, akkor az ütemező kénytelen a két ág futtatását sorosítani. Tehát először az if ág fut le, úgy hogy közben a warp összes olyan szála, amelyre az else teljesül letiltásra kerül, majd következik az else ág.



2. ÁBRA MEMÓRIAOLVASÁSOK, ÉS FUTTATÁS ÁTLAPOLÓDÁSA[7]

Ezen warpok ütemezése általában aszerint történik, hogy épp melyik warp melyik utasításának áll rendelkezésre az összes bemeneti paramétere, ha ezek nem állnak készen, akkor kénytelenek várakozni. Általában ez a várakozás a memóriaolvasások eredményeire történik. Az esetek jelentős részében, a memória műveletek jelentik a legidőigényesebb műveletet, a későbbiekben az lesz a célunk, hogy ezek számát visszaszorítsuk.

Az architektúra meghatározó tulajdonsága, hogy ezen warpok kontextus váltását rendkívül gyorsan végre tudja a hardver hajtani. Tehát amíg épp egy warp egy memória olvasást kezdeményezett, és ennek eredményére várakozva „blokkolódott", addig az ütemező egy másik warpot tud elindítani, ezen tulajdonság érdekes következménye, hogy a memóriaolvasások ideje egy jelentős korlát, amit el kell fogadnunk. Azonban rajtunk programozókon múlik, hogy hány szálat-warp-ot hozunk létre, ha ezt az értéket megfelelően magasra választjuk, akkor a ***2. ábrán*** látható módon elég munkát adunk a GPU processzorainak, ahhoz, hogy amíg az olvasás művelet lezajlik, az további olvasásokat kezdeményezhessen és egyéb műveleteket hajthasson végre. Így

elérhető, hogy a feldolgozás elfedje az olvasások jelentős késleltetését. Tehát, mire az első olvasás eredményét feldolgoztuk, addigra jó eséllyel lezajlik a második olvasás, és így tovább.

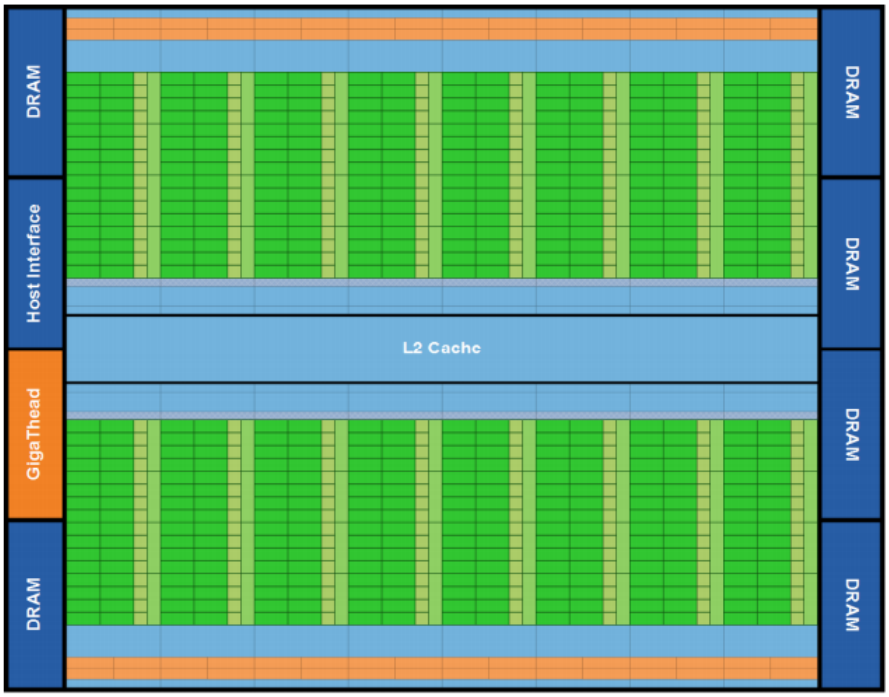
2.2 COMPUTE CAPABILITY, KÁRTYÁK KÜLÖNBSÉGEI

Az Nvidia azért, hogy a CUDA alkalmas kártyákat képességeik szerint könnyen be tudja határolni, ezeket egy Compute Capability nevű értékkel címkézi. Ezen érték két részből áll egy „*major*" és egy „*minor*" részből, előbbinél jelenleg két érték lehetséges, ha a kártya az „új" (2010 nyarán kiadott), Fermi architektúrán alapul, akkor ez az érték ***2***-es, ha még az előtti, akkor ***1***es. [4]

A minor értékek főleg a Fermi előtti architektúránál jelentősek, itt a létező 4 típust 2 csoportra lehet osztani:

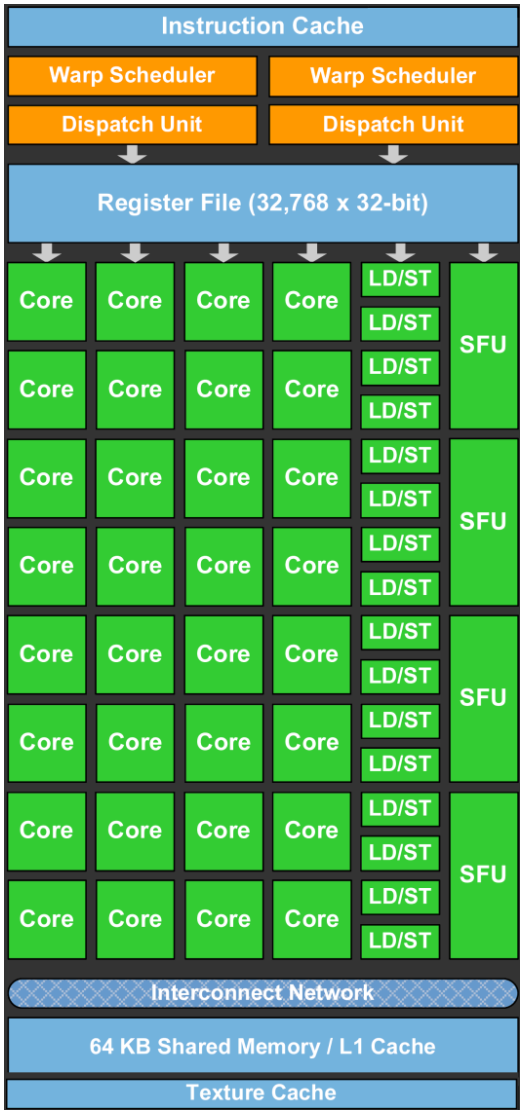
Az **1.0** és **1.1** között csupán annyi a különbség, hogy **1.1**-ben már megjelentek az atomi műveletek. Az első ilyen kártya 2006 végén jelent meg, hamarabb, mint az első cuda fejlesztői eszköz. Mint egy későbbi példán látszani fog ezen csoportra való optimalizálás meglehetősen nehéz, komoly tervezői munkát igényel és mivel ezen kártyák ma már nem rendelkeznek olyan jelentős számítási kapacitással, ezekkel szemben nem a teljes optimalitás, hanem a ***biztos futás*** lesz az elsődleges követelmény.

2009 elején jelent meg az első **1.2**-es kártya, én is ilyennel rendelkezem, ez még napjainkban is jelentős sebességfölényt mutathat akár a leggyorsabb asztali processzorokkal szemben is, így a Fermi-n kívül ez lesz az optimalizáció egyik legfontosabb platformja. Ebbe a csoportba tartozik még az **1.3**, amit főként az különböztet meg az **1.2**-től, hogy képes dupla pontosságú lebegőpontos műveleteket is végezni, habár ezek teljesítménye, még nem túl magas.



3. ÁBRA A FERMI ARCHITEKTÚRA TELJES BLOKKVÁZLATA[8]

A ***3. ábrán*** látható a Fermi teljes blokkvázlata, most ennek legérdekesebb részén a számításokat végző Streaming Multiprocesszorokon (***4. ábra***) keresztül mutatom be az előbbi két csoport és a Fermi közötti legfőbb különbségeket:



4. ÁBRA FERMI STREAMING MULTIPROCESSZOR BLOKKVÁZLATA[8]

2.3 STREAMING MULTIPROCESSZOROK

2.3.1 STREAMING PROCESSZOROK (CORE)

A legészrevehetőbbek az ***4. ábrán*** a processzor magok, ezeknek a száma 32 vagy 48, a korábbi architektúrákon ez a szám 8 volt. Fentről lefelé haladva látható, hogy itt a korábbiakkal ellentétben már 2 warp ütemező van, többek között ennek is köszönhetően akár egyszerre több kernelt is futtathatunk, ennek akkor van előnye, ha relatív kisméretű programjaink vannak, amik nem használják ki teljesen a GPU-t, akkor lehetőségünk van ezeket egyszerre lefuttatni.

2.3.2 REGISZTEREK

A következő egység az egyik legfontosabb, ezek a regiszterek, mint látható 32*1024 darab 32 bites regiszter található multiprocesszoronként, kétszer annyi, mint az **(1.2,1.3)** párban, és négyszer annyi, mint az **(1.0,1.1)** csoportban.

2.3.3 MEGOSZTOTT MEMÓRIA, KÉTSZINTŰ CACHE-ELÉS

A ***4. ábrán*** még két nagyon fontos egység látható, abból az egyik összetett. Ez a megosztott memória, aminek egy 64 KB méretű cache-t kell megosztania a Fermi egyik legnagyobb újításával, az L1 cache-el. Nagy módosítás a korábbiakhoz képest, hogy ezen architektúrán a globális memóriaolvasások két szinten (L1,L2) cache-elve vannak. Ennek köszönhetően a szálak először az L1-ből próbálnak olvasni, majd sikertelenség esetén L2-ből és végül, ha az előző kettőben nem találták meg a szükséges adatokat, akkor a globális memóriából. Mivel az L1 és a megosztott memória, egy cache-re képződik le így, ezek közötti arány változtatható tetszőlegesen a programozó által 16KB és 48 KB arányokban.[9]

Itt mutatom be a megosztott memóriát részletesen, ami általában a kernelek optimalizációjának egyik legfontosabb eleme.

A teljesítményét legnagyobb mértékben befolyásoló tényezője, az hogy bankokból áll. A Fermi előtti architektúrán a mérete 16 KB volt, és ehhez 16 bank társult, tehát egy bank mérete 1 KB volt. Abban az esetben, ha egyszerre egy warp-nak több szála is ugyanazt a bankot szeretné olvasni vagy írni, akkor úgynevezett bank-konflikust lép fel. Ebben az esetben a kérések „beérkezése" szerinti sorrendben sorosítás történik.

Tehát egy bank egy időben csak egy szálat tud kiszolgálni. Annak érdekében, hogy a 32 szálból álló warpok, optimális elrendezés esetén ne okozzanak konfliktust, a warp-ok két részletben hajtják végre a megosztott memória műveleteket, először az első 16 szál, majd a második 16 szál, így ezek között nem léphet fel konfliktus. Meg kell jegyezni, azon fontos és nem triviális tulajdonságot, hogy különböző warp-ok között nem léphet fel konfliktus.

Fermi esetén komoly változások történtek, a megosztott memória maximális mérete 48 KB lett és ez 32 bankra oszlik, itt viszont figyelni kell arra, hogy ha a warpok első és második 16 szála ugyanazt a bankot próbálja olvasni, akkor felléphet konfliktus. Fontos megjegyezni, hogy ezen bankok architektúrától függetlenül 32 bites szavakból állnak, ha egy warp több szála egy bankon belül ugyanazt a szót szeretné kiolvasni, akkor az broadcastolódik, így nem lép fel konfliktus.

2.3.4 WARPOK FUTTATÁSA A STREAMING MULTIPROCESSZOROKON

A három Compute Capability csoport a Streaming Multiprocesszorokon futtatható maximális szálak számában is különbözik egymástól. Ezen érték értelemszerűen Fermin a legmagasabb, 1536. Azért fontos ezen értékkel tisztában lennünk, mert ez jelenti a felső korlátot arra, hogy egy multiprocesszoron egyszerre hány warp futtatható.

2.3.4.1 OCCUPANCY

Az ehhez kötődő legfontosabb fogalom az ***occupancy***,*[10]* ami azt mutatja meg, hogy egy multiprocesszoron a maximálisan aktív warpok számnak hány százaléka ténylegesen aktív. Aktív alatt értve azt, hogy ha épp nem várakozik akkor az ütemező futtathatja. Tehát ez az érték a kernelek egyik fő jellemzője, amit az alábbi 3 tényező és azok egymásra hatása befolyásol:

- ***A blokkonkénti szálak száma***
- ***A szálankénti regiszterek száma***
- ***A blokkonként használt megosztott memória mérete***

A blokkonkénti szálak számát a programozó feladata meghatározni, erre a korlát Fermin 1024-re nőtt, előtte 512 volt. Architektúrától függetlenül maximum 8 blokk lehet aktív egy

multiprocesszoron, így az ***occupancy*** értéke már akkor is 100% alá csökkenhet, ha például 1 blokk 32 szálból áll, tehát 1 warp-ból, így 8 blokk futhat egyszerre, de így is csak 8 warp jöhet létre. A Fermi maximális 1536 szálból következik, hogy a warp-ok maximális száma 48, ebből csak 8at tudtunk létrehozni, így az ***occupancy*** 17% lett.

Mint korábban írtam Fermin 32*1024 regiszter található minden multiprocesszorban, ami egy meglehetősen nagy értéknek tűnik, de ugyebár a multiprocesszoron futó összes blokk összes szálának összes regiszterének bele kell férnie ebbe a korlátba, ha nem fér bele úgynevezett ***register spilling*** léphet fel, erről bővebben a következő alpontban írok.

Fontos tudni, hogy regiszterek 32 bit szélesek, a fordító ezekbe próbálja elhelyezni az összes változót, így növelve a teljesítményt, hiszen ezeket néhány órajel alatt lehet olvasni és írni, azokat, amikre már nincs szükség próbálja kioptimalizálja.

Egy példa, amin jól látható, hogy a regiszterhasználat hogy változtathat az ***occupancy***-n:

Van egy kernelünk, amihez blokkonként 256 szálat hoztunk létre, szálanként 42 regisztert használ. Tehát egy blokknak 10752 regiszterre van szüksége, ami azt jelenti, hogy az elérhető 32768 regiszterben ez 3szor fér el, tehát 3 blokk jöhet létre. Mivel tudjuk 256/32=8 warp van egy blokkban, így összesen egy multiprocesszoron 24 warp futhat, ami a maximális 48 warp-nak az ***50%***-a.

Viszont, ha eggyel több változót használunk, amit a compiler nem tud optimalizálni, tehát egy már meglévő, de nem használt regiszterben elhelyezni, akkor eggyel több regiszterre lesz szükségünk. Ekkor a blokk igénye 11008 regiszter, így már csak 2 blokk jöhet létre ennek köszönhetően az ***occupancy*** 16/48=***33%***-ra csökken.

Ebből kiderül, hogy egy blokknak vagy az összes warp-ja létrejön, vagy egyik sem.

Ezen példa jól bemutatja, hogy a programozói oldalon kicsinek tűnő változtatások milyen jelentős következményekkel járhatnak.

2.3.4.2 REGISZTER SPILLING

Annak érdekében, hogy ezen „negatív hatások" ne befolyásolják ilyen drasztikusan a teljesítményt, illetve hogy ha több regisztert szeretnénk használni, mint ami elérhető, a hardver úgynevezett ***regiszter spilling***-hez folyamodik. Ez azt jelenti, hogy egy bizonyos regiszterszám felett már nem a regiszterekben allokál helyet, hanem a szálakhoz tartozó lokális memóriában, így elkerülhető az ***occupancy*** csökkenés. Habár a lokális memória a globális memória része, így két nagyságrenddel lassabban lesznek elérhetőek az oda „kihelyezett" változók, viszont több warp-ot tudunk létrehozni, így az olvasások késleltetését jobban át tudjuk lapolni a feldolgozással, a korábban említett módon.

Az ***occupancy***-t befolyásoló harmadik tényező a felhasznált megosztott memória mérete. Ezt statikusan illetve dinamikusan lehet allokálni.

Az ***occupancy***-t ez esetben az határozza meg, ha más tényező nem limitálja, hogy a rendelkezésre álló 16 vagy 48 KB memóriában hány blokk igénye fér el.

2.3.4.3 SZÁL SZINKRONIZÁCIÓ JELENTŐSÉGE

Annyival összetettebb a helyzet a regiszterekhez képest, hogy általában a megosztott memóriába való betöltésnél más szál elrendezést használunk, mint az az utáni feldolgozásnál, ennek köszönhetően előfordulhatna olyan állapot, hogy egy szál betöltötte az adatokat, és mikor

a feldolgozási fázisba lépne, olyan adatot kellene felhasználnia, amit egy olyan szálnak kellett volna betöltenie, ami még lehet, hogy le sem futott.

Ezen szituáció elkerülésére szolgál a ***__syncthreads()*** függvény, ennek hatására, a blokk összes warp-jának minden utasítását le kell futtatnia a szinkronizációs pontig. Azok a warp-ok, amik elérték ezt a pontot blokkolódnak, kénytelenek várakozni, így a ténylegesen futtatható warp-ok száma ideiglenesen csökken, ekkor nő meg annak a jelentősége, hogy egy multiprocesszoron hány blokk aktív éppen. Hiszen így amíg az egyik blokk a szinkronizáció ja miatt egyre kevesebb warp-ja aktív, addig az ütemezőnek lehetősége van más blokkok warp-jainak futtatásával elfedni a globális memória késleltetéseit, tehát a párhuzamosság mértéke nem csökken.

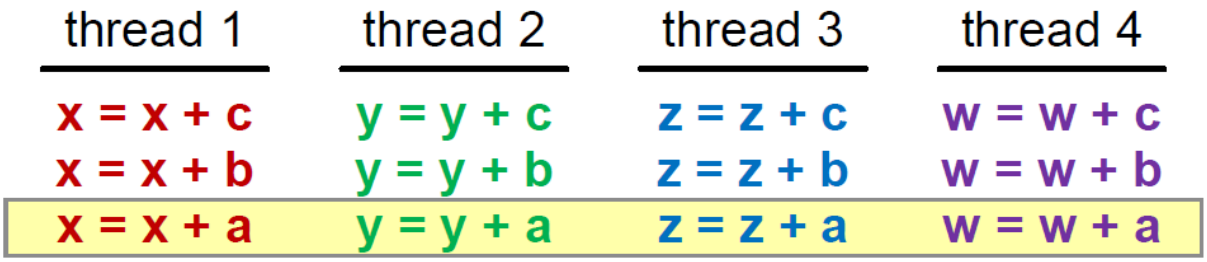
2.3.4.4 OCCUPANCY HATÁSA A TELJESÍTMÉNYRE

Nem ejtettem még szót az ***occupancy*** teljesítményre gyakorolt hatásáról, ez nem véletlen, hiszen ez egy meglehetősen összetett kérdés.

Az általános nézőpont az, hogy ha az ***occupancy*** alacsony, akkor a globális memória olvasások eredményére várva, nincs elég warp-unk, amit be tudunk ütemezni, tehát a késleltetést nem tudjuk „elrejteni" más warp-ok feldolgozásával. Emellett a másik véglet az, hogy egy bizonyos szintnél nem érdemes tovább növelni az ***occupancy***-t, mert afelett a szint felett, ahol már elfedtük a késleltetéseket, a növelése általában nem okoz teljesítménynövekedést.

A jelenlegi összes Nvidia[*4*][*12*] segédanyag ezt a nézőpontot támogatja, azonban ez nem mindig ad helyes képet, tehát lehet nagyon alacsony ***occupancy*** mellett nagy teljesítményt elérni, ennek megértéséhez be kell mutatnom a GPU-n elérhető két féle párhuzamosságot.[*11*]

Az egyik a ***5. ábrán*** is látható ***szál szintű*** párhuzamosság, mint látható a szálak párhuzamosan végrehajtanak egy műveletet, ha az „**a**" változó a globális memóriában tárolódik, akkor meg kell várniuk, amíg onnan betöltődik, majd végrehajthatnak egy műveletet.

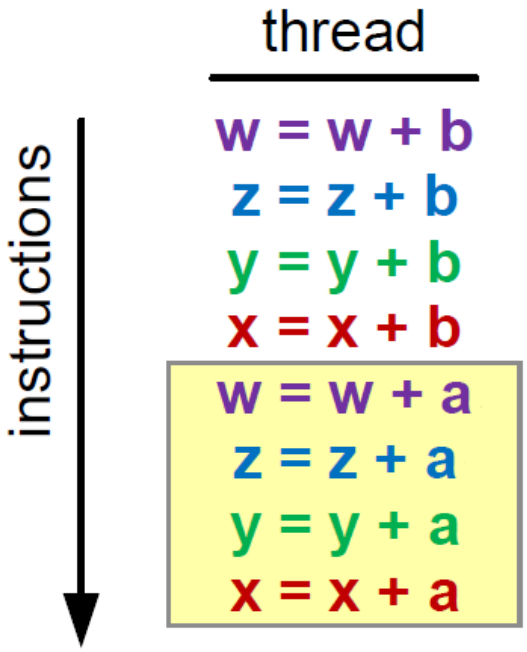


5. ÁBRA SZÁL SZINTŰ PÁRHUZAMOSSÁG[11]

A második esetről a CUDA programozók általában nem tudnak, és így nem is figyelnek a tervezés során erre, hogy lehetőség van utasítás szintű párhuzamosságra is. Erre egy példa ***6. ábrán*** látható. Ilyenkor a compiler felismeri a változók függőségeit, mint például az előző ábrán a második sorban. Ha feltételezzük, hogy „**b**" egy regiszterben tárolt érték, és a harmadik sorban „**a**" szintén regiszter, akkor a 2 utasítás a függőség miatt nem hajtható végre, ugyanis ahhoz, hogy „**x**"-et „**a**"-val tudjuk növelni, meg kell várnunk amíg a **b**-vel való növelés végrehajtódott, és eredménye láthatóvá vált, ez körülbelül 24 órajelnyi időt vesz igénybe.[*11*]

Ezzel ellentétben az ***6. ábrán*** látható, hogy a bekeretezett négy művelet között nincs függőség, tehát az értékének birtokában ugyanabból a warp-ból négy utasítás is végrehajthatóvá válik. Így látható, hogy ugyanazt a feladatot, amire eddig négy szálra volt szükségünk, most egyetlen szállal valósítottuk meg, ennek az ára persze az volt, hogy nőtt a regiszterigény. Ezáltal

csökken az ***occupancy***, de ebben az esetben ez nem olyan zavaró, hiszen a szálaink több feladatot láthatnak el, tehát ilyen módon a meglévő erőforrásokat alternatív módon is eloszthatjuk. Ilyenkor például a globális memória olvasások eredményeire várva nem egy másik warp-ot futtatunk, hanem ugyan annak több más utasítást. Tehát kijelenthető, magas utasítás szintű párhuzamossággal kis ***occupancy*** mellett is érhető el nagy teljesítmény.



6. ÁBRA UTASÍTÁS SZINTŰ PÁRHUZAMOSSÁG[11]

Az ***occupancy*** növelésével, illetve az utasítás szintű párhuzamossággal tehát elsősorban a késleltetés elrejtésére voltunk hatással, azonban a késleltetés csak egy tényező azon három közül, amik a kernel teljesítményét meghatározzák. A másik kettő a GPU által egységnyi idő alatt végrehajtható műveletek száma, illetve a globális memória sávszélessége, ezekről röviden a következő pontban írok.

2.3.5 A KERNELEKET TELJESÍTMÉNYÉT LIMITÁLÓ TÉNYEZŐK

Mint láthattuk, magas ***occupancy***-vel, vagy magas utasítás szintű párhuzamossággal jelentősen javítható egy kernel teljesítménye, azonban ez csak annak köszönhető, hogy elfedtük a globális memória késleltetését. Ha ezt megtettük a kernel futását még mindig két nagyon fontos tényező limitálhatja.

Ezek közül az egyik, a globális memória sávszélessége, ha ennél több adatot szeretnénk kiolvasni egységnyi idő alatt, akkor a busz nem fogja tudni kiszolgálni a multiprocesszorokat, így csökken a teljesítmény.

A másik az egységnyi idő alatt elvégezhető műveletek száma, ez akkor jelent limitációt, ha a globális memóriából kiolvasott adatokat a kernel nem tudja kellően gyorsan feldolgozni, mert a feldolgozás annyira számításigényes, hogy ahhoz nem tud elegendő utasítást végrehajtani. Ebben az esetben a globális memória sávszélességének csupán töredékét használja a kernel, mert a feldolgozás annyira leterheli a multiprocesszorokat, hogy azok nem tudnak elegendő memória műveletet kezdeményezni, ahhoz, hogy kellően kihasználják a memória sávszélességét.

2.3.6 A GLOBÁLIS MEMÓRIA JELLEMZŐI

Korábban már esett szó a globális memória olvasások és írások kritikus sebességéről, amik meglehetősen nagy hatással vannak teljesítményre, ezen pontban ezek tulajdonságait mutatom be.

Az egyik legfontosabb információ, hogy a GPU 32, 64 és 128 bájtos memóriaterületeket tud csak olvasni, kisebbet sajnos nem. Kedvezőtlen esetben a kiolvasott adatmennyiség jelentős része haszontalan így eldobásra kerül. Jelentős fegyver a programozó kezében, amivel tud harcolni ez ellen, hogy a compiler képes az egy időben esedékes kis (egy warp-on belül történő) memóriaolvasásokat összevonni egy nagy olvasássá, ha ez lehetséges.

Az optimalizáció szempontjából kulcsfontosságú, hogy a compiler-nek ezt a tevékenységét a programozó mennyire támogatja.

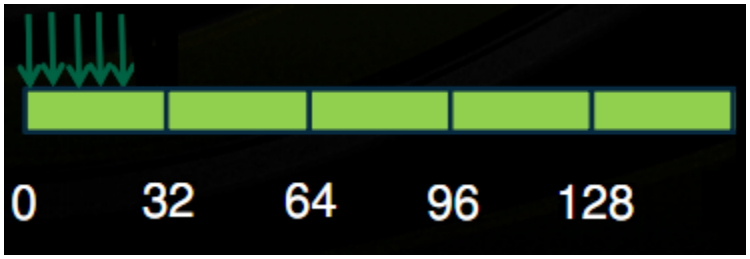
A területhez kapcsolódó fogalom az úgynevezett ***coalesced*** olvasás, ilyen akkor jöhet létre, ha a warp-on belüli szálak egy időben, folytonos memóriaterületet olvasnak, azon olvasásokat, amikre ez nem teljesül ***uncoalesced*** olvasásnak nevezzük.

Erre látható egy példa a ***7. ábrán***, itt a cél 5 *float* változó kiolvasása, egy *float* mérete 4 bájt. Azonban tudjuk, hogy a legkisebb kiolvasható terület 32 bájt, látható a képen, hogy a kiolvasandó *float*-ok 32 bájt távolságra vannak egymástól. Ez akár jelenthet egy 1 dimenzióba leképzett 2 dimenziós tömböt, amiből minden sorának első elemét szeretnénk kiolvasni. Ebben az esetben a GPU kénytelen 5 db 32 bájtos olvasást végrehajtani, majd a felesleges kiolvasott értékeket eldobni, tehát a teljes kiolvasott memóriaterületnek csupán a ***12,5 %*** volt hasznos. Mivel egy olvasás ***400-800*** órajel késleltetéssel jár így ez hatalmas pazarlás.

7. ÁBRA UNCOALESCED GLOBÁLIS MEMÓRIA OLVASÁS[4]

Ezzel ellentétben, ha előre tudjuk, hogy a kernelünk warpok szintjén egy időben mely területet fogja olvasni, akkor lehetőségünk van ezeket az értékeke egymáshoz közel elhelyezni, gondolva arra, hogy így lehetőség van ezeket egy olvasássá összevonni. A ***8. ábrán*** látható esetben szintén 5 float értéket szeretnénk kiolvasni, de annak köszönhetően, hogy ezek egymás mellett találhatóak egyetlen 32 bájtos olvasás elegendő.

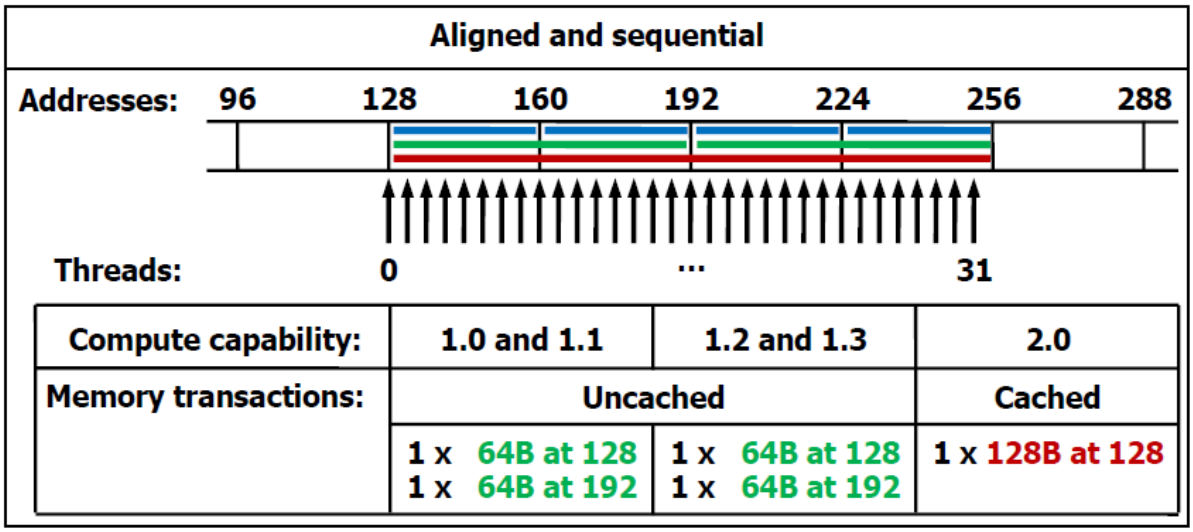
Emiatt, a kiolvasott értékek ***62,5 %*** volt hasznos. Ez az érték természetesen jóval magasabb is lehet, de ezen egyszerű példa jól bemutatja, hogy mekkora különbségeket lehet elérni, ha a programozó a tervezés során odafigyel arra, hogy az olvasások ***coalesced*** módon történjenek.



8. ÁBRA COALESCED GLOBÁLIS MEMÓRIA OLVASÁS

A ***coalesced*** olvasások bevezetése után néhány konkrét példán mutatom be, hogy az egyes Compute Capability csoportok, miként tudják összevonni a kis olvasásokat nagyobbakká, ezzel is javítva a teljesítményt. Ahogy már korábban írtam, az **(1.0,1.1)**, csoport optimalizációja meglehetősen sok munkát igényel, ez annak köszönhető, hogy többek között ezen a területen is komoly hiányosságokkal rendelkezik modernebb társaihoz képest, az elkövetkező három példa ezt jól prezentálja. Az olvasásoknál float-okról van szó, hogy egységesen lehessen ábrázolni.

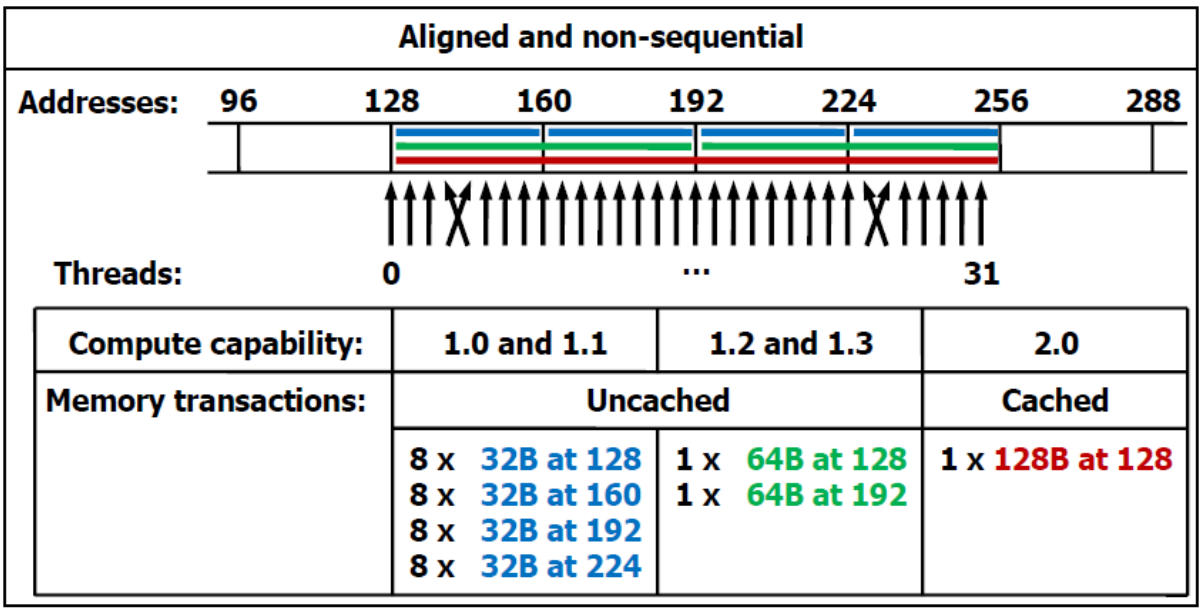
A ***9. ábrán***, a lehető legoptimálisabb esetről van szó, mikor a kiolvasandó területről a warp szálai az azonosítójuknak (jelen esetben 0-31) megfelelő értéket akarják kiolvasni. Látható, hogy ebben az esetben még az (1.0,1.1) és az (1.2,1.3) csoport azonosan teljesít, de egyik se tudja összevonni ezt az olvasást egy nagy 128 bájtos olvasássá.



9. ÁBRA TÖKÉLETESEN COALESCED OLVASÁS[4]

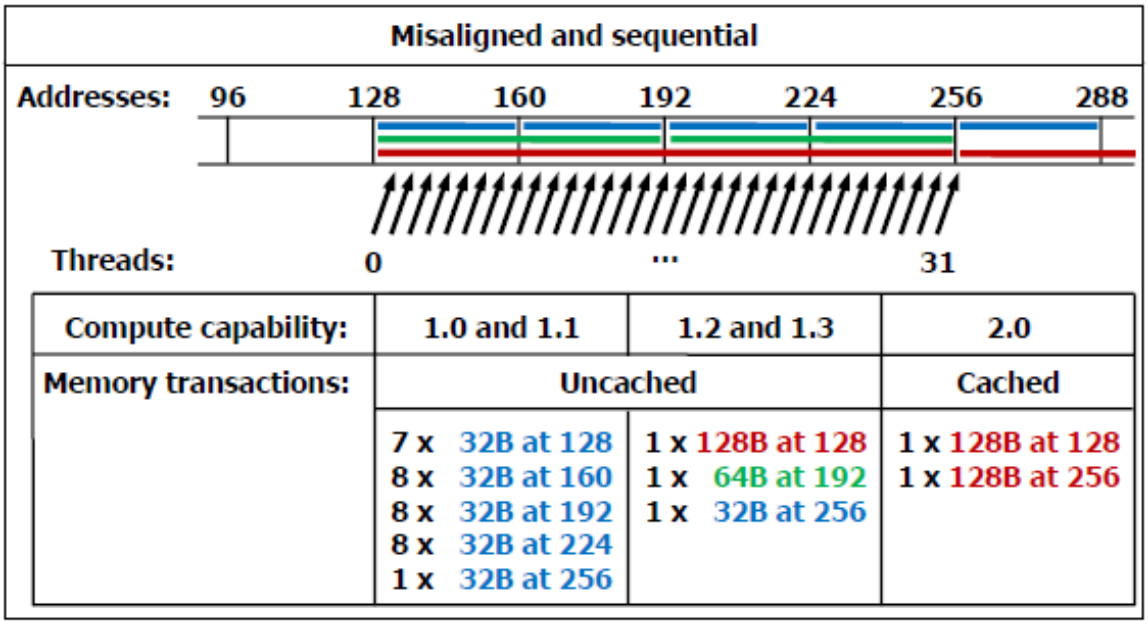
Fontos jellemzője a Ferminek, mint korábban írtam, hogy a globális memória olvasások két szinten cache-elve vannak, ami ezáltal jóval gyorsabb, mint a korábbi architektúrák, ezt részletesen később elemzem.

A ***10. ábra***, azt az esetet mutatja be, amikor a globális memóriában tárolt adatok sorrendje nem egyezik, meg azon szálak azonosítóinak sorrendjével, aminek szüksége van az adott kiolvasott értékre, és a memóriaterület a szálak azonosítóival van indexelve. Ekkor kiolvasás után a szálak nem tudnak sorrendben hozzáférni az adatokhoz, meg kell cserélni a sorrendet, ez meglehetősen egyszerű problémának tűnik, de az **(1.0,1.1)** csoport, mint látható ezt már nem tudja optimálisan végrehajtani, így mint korábban leírtam a lehető legrosszabb eset következik be, minden lebegőpontos értéket egyesével olvas ki, minden olvasás 32 bájtos, és a fennmaradó 28 bájtot eldobja.



10. ÁBRA COALESCED "NON-SEQUENTIAL" OLVASÁS[4]

A ***11. ábrán*** látható esetet a legegyszerűbben úgy lehet jellemezni, ha egy a globális memóriában lévő t tömbből **t[szál_azonosító+n]** módon olvasunk ki adatokat, ebben az esetben ***n=1***. Az ilyen megoldások előfordulása nagyon gyakori, ezért is kellemetlen, hogy az **(1.0,1.1)** csoport ezzel sem tud relatív optimálisan megbirkózni.

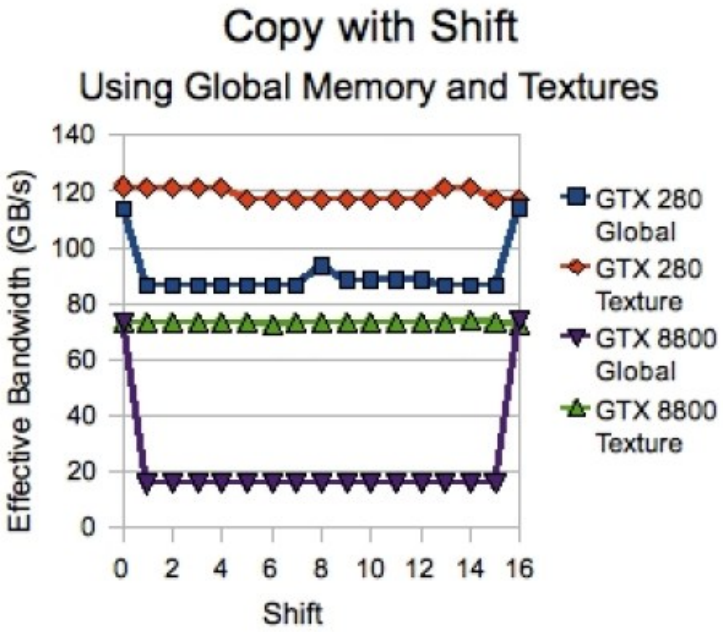


11. ÁBRA COALESCED MISALIGNED OLVASÁS[4]

2.3.7 ALTERNATÍV MEMÓRIA TÍPUSOK

Bizonyos alkalmazási területeken nincs lehetőségünk az adatokat **coalesced** olvasásra optimálisan rendezni. Ekkor lehetőségünk van alkalmazni a textúra memóriát. A textúrák a globális memóriában tárolódnak, de multiprocesszoronként 6-8 KB-nyi cache-el van gyorsítva a kiolvasásuk. A textúrákat csak a kernel indítása előtt lehet írni, futási időben csak olvashatóak.

A mérések szerint a textúra referenciákon keresztül történő globális memória elérések a Fermi előtti Compute Capability-ken, általában gyorsabb, mint a **coalesced** memória olvasások sebessége.



12. ÁBRA TEXTÚRA MEMÓRIA ÉS GLOBÁLIS MEMÓRIA SÁVSZÉLESSÉG TESZJE[12]

A **12. ábrán,** a **11. ábrán** megfigyelhető eltolás negatív hatását láthatjuk, és ennek áthidalását textúrával.

A GTX 280 kártya, ***1.3***-as, a GTX 8800 pedig ***1.0***-ás Compute Capability-vel rendelkezik. Látható, hogy az eltolás a 8800-on jóval komolyabb teljesítmény csökkenést okoz, és hogy a textúra memória cache-ének alkalmazásával ez relatív kis veszteséggel áthidalható.

Fermi esetén ahol a globális memória olvasások két szinten cache-elve vannak, általános esetben nem tud a textúra memória jelentős teljesítményfölényt felmutatni, azonban ez függ az alkalmazástól.

Az egyetlen memóriatípus, amiről idáig nem volt szó a konstans memória, ez hasonlít a textúra memóriára olyan szempontból, hogy szintén a globális memória egy elkülönített területe, és ez is cache-elve van, ennek mérete multiprocesszoronként 8 KB.

A maximális mérete globálisan 64 KB architektúrától függetlenül. Olyan szempontból pedig a megosztott memóriára hasonlít, hogy bankokra van osztva, és *broadcast*-olásra van optimalizálva, a legnagyobb teljesítményt akkor érhetjük el rajta, ha egy warp összes szála, a konstans memória azonos szavára hivatkozik.

2.4 TOVÁBBI CUDA OPTIMALIZÁCIÓS ESZKÖZÖK

2.4.1 STREAMEK

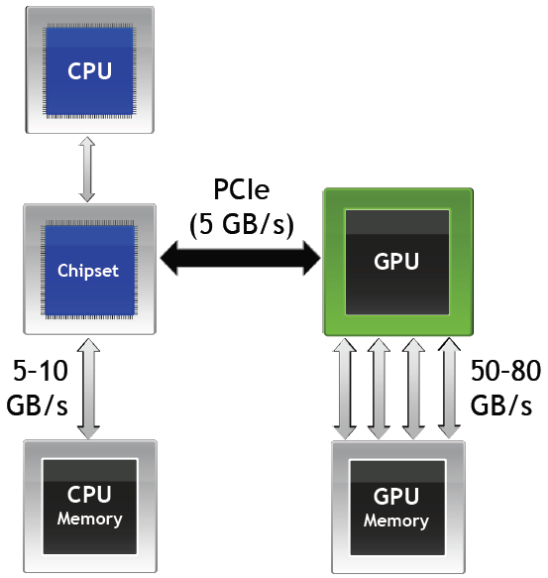
Ennek bemutatása előtt le kell tisztázni, hogy a ***CPU<->GPU*** másolások alapvetően szinkron módon történnek, tehát amikor a CPU kezdeményez egy ilyen másolást, akkor a CPU-n futó kód blokkolódik egészen a másolás végéig. Ezzel ellentétben a kernelek futtatása aszinkron módon történik, tehát a kernel indítása után az irányítás azonnal visszatér a CPU-hoz, így lehetőség van a CPU-n a GPU-val párhuzamosan feladatokat végezni.

Ez azért fontos, mert habár a GPU elméleti számítási kapacitása akár százszorosa is lehet a CPU-énak, hatalmas pazarlás az, ha a CPU-n futó kód, a GPU futtatás közben csak várakozik. A ***stream***-ek GPU-n végezendő műveletek sorából állnak, ezek lehetnek memóriaműveletek, illetve kernel futtatások. Ha egy ***stream*** futtatását kezdeményezzük, akkor annak műveletei a CPU-hoz viszonyítva aszinkron módon kerülnek futtatásra. Az *N*. művelet csak akkor indul el, ha az *(N-1)*. már lefutott. Így a CPU-n eközben a GPU-tól függetlenül tudunk dolgozni, szinkronizációt pedig úgynevezett ***esemény***ek segítségével végezhetünk.

Komoly lehetőség, hogy az **1.2**-es Compute Capability-től felfelé a kártyák képesek egy időben másolni és kernelt futtatni. A Fermis kártyák pedig két irányú másolásra és kernel futtatásra is képesek párhuzamosan. Tehát ennek köszönhetően, amíg az ***N.*** adathalmazon dolgozik a GPU, addig az ***(N-1).*** munka eredményét visszamásolhatjuk a CPU-hoz, illetve az ***(N+1).*** adathalmazt előkészíthetjük, majd másolhatjuk a GPU-ra. Ezen technikát később használni fogom.

2.4.2 PAGE-LOCKED MEMÓRIA

A CPU<->GPU másolások aszinkron végrehajtásának elemzése után ezek sebességét leginkább befolyásoló fogalmakat mutatom be.



13. ÁBRA PC SÁVSZÉLESSÉG ADATOK 2009-BŐL[13]

Habár a ***13. ábrán*** látható sebességek csak a 2009-es mért adatokat tükrözik, de azok aránya még napjainkban is helytálló. Látható, hogy a szűk keresztmetszet a PCI express busz sávszélessége. De van egy még ennél is komolyabb korlát, hiszen ezek a sebességek, abban az

esetben állnak fenn, ha az általunk másolni kívánt memóriaterület lapjai ténylegesen a CPU memóriájában találhatóak, tehát az operációs rendszer virtuális memória kezelője nem helyezte ki azokat a lapokat a háttértárra.

Ha ez megtörtént, akkor először onnan kell betölteni a memóriába, majd csak ezután történhet meg a GPU memóriába való másolás, tehát ilyen esetben még a PCI express busz sávszélességét se tudjuk megfelelően kihasználni. Ennek elkerülése érdekében a CUDA tervezői lehetőséget biztosítottak arra, hogy úgynevezett ***page-locked***, vagy más néven ***pinned*** memóriát foglaljunk.

Ezen memóriaterület, ténylegesen a CPU memóriájába képződik le, úgy hogy onnan felszabadításáig ne lehessen kihelyezni a háttértárra, emellett még fix memóriacímet is kap, tehát a GPU a CPU-val való „egyeztetés" nélkül másolhatja és írhatja ezen területeket, ezzel terhet vehetünk le a CPU-ról is.

A ***page-locked*** memória tehát általában jelentősen lecsökkenti a másolási időket, de ez sajnos nem minden esetben igaz, mert minden rendszernek megvan a saját korlátja, tehát ha túl sok ilyen memóriát foglalunk, egy bizonyos szint felett már nem jut elegendő memória az operációs rendszernek, és az nem tudja hatékonyan ellátni feladatait.

2.5 A CUDA ALKALMAZÁSOK FORDÍTÁSÁNAK LEHETŐSÉGEI:

A CUDA programok fordítója az NVCC, ez először a CPU-n futó kódot elkülöníti, majd átadja az adott operációs rendszer alapértelmezett C fordítójának. Mivel ez a fordítás könnyebb része így ez a CPU programokhoz hasonlóan történik, ennél sokkal érdekesebb a GPU kód fordítása.

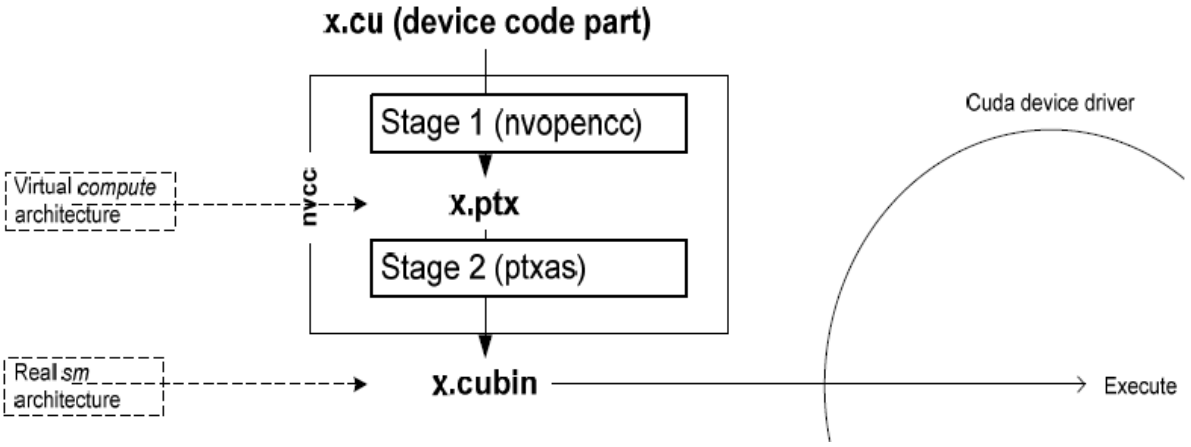
Ennek első lépése CU forrásfájlok a PTX-be történő fordítása. A PTX a Paralell Thread Execution, egy ember számára is olvasható assembly nyelv, aminek célja, hogy a forráskód, egy olyan köztes fordítási állapotba kerüljön, ahonnan már „Just In Time" módon lehessen fordítani a tényleges futtató architektúrára.[14]

A kompatibilitás megőrzése érdekében, a PTX csak egy „virtuális architektúrára" fordul le, így ez a kód még optimalizáció előtt áll, hiszen, addig nem tudunk tényleges optimalizációt végrehajtani, amíg nem tudjuk, hogy mely architektúrán akarjuk a kódunkat futtatni.

PTX mellett lehetőségünk van CUBIN-ba fordítani, mint ahogy a ***14. ábrán*** is látható, ez már gépi kód, ami egy tényleges architektúrára (~Compute Capability csoportra) van optimalizálva, ennek köszönhetően a kompatibilitása jóval kisebb, mint a PTX fájloké.

A CUBIN fájlok se előre se visszafelé nem kompatibilisek más architektúrával, ennek az oka az, hogy például a Fermi a globális memóriájának 2 szintű cache-elése miatt, teljesen más parancsokat kell végrehajtani, mint ***1.0***-án.

Tehát a jövőbeli kompatibilitás miatt érdemes a PTX-et előnyben részesíteni, még akkor is ha a JIT fordítás miatt, első futtatásra lassabban is töltődik be, mint a CUBIN.



14. ÁBRA CUDA FORDÍTÁS MENETE[14]

Miután a PTX vagy CUBIN fájlunk elkészült, felmerül bennünk a kérdés, mégis hogy tudjuk ezeket futtatni, hiszen nincs CPU-n futó komponensünk, ennek megértése érdekében röviden bemutatnom a CUDA két rendelkezésre álló API-ját.

Ezek a Runtime és a Driver API. A Runtime API-t felhasználó barátiabb, tehát nincs szükség külön inicializációkra, amiket a Dirver API esetén meg kell tenni. A Runtime API összes funkciója megtalálható a Driver API-ban, de ebben talán az egyik legfontosabb kiegészítés, hogy lehetőségünk van modulokat betölteni. Ezen modulok többek között lehetnek a fent említett PTX vagy CUBIN fájlok. Miután ezeket betöltöttük egy modulba, lehetőségünk van a függvényeket egyesével kiemelni belőlük ezek után már futtathatóak is.

3. LÉTEZŐ CUDA K-NN MEGOLDÁSOK, PUBLIKÁCIÓK

3.1 K LEGKÖZELEBBI SZOMSZÉD ALGORITMUS

A k legközelebbi szomszéd (k-NN) algoritmus egy széles körben alkalmazott osztályozási, gépi tanulási és minta felismerési módszer. Sajnos a magas számításigénye miatt, alkalmazási területeinek jelentős részén nem tudunk vele kielégítő teljesítményt elérni. A lusta tanuló algoritmusok csoportjába tartozik, tehát nem épít modellt.

Lényegében arra alapoz, hogy ha két objektum attribútumai nagyon hasonlóak, akkor jól közelíthetjük az egyik ismeretlen attribútumait, a másik ismertjei alapján. Ezt a hasonlóságot, egy távolságfüggvénnyel mérjük. A futtatáshoz szükségünk van egy tanító és egy teszt halmazra. Célunk, hogy a tanító halmazban rejlő tudás alapján kitaláljuk a teszt halmaz elemeinek ismeretlen tulajdonságait. Ehhez három lépésre van szükségünk:

1. Teljes távolságmátrix számítása a tanító és a tesztelő halmaz között.
2. A távolságok alapján megtalálni minden teszt objektumhoz a hozzá legközelebb álló k tanító objektumot. Ehhez általában teljes, vagy részleges rendezésre van szükség.
3. A megtalált k objektum segítségével eldönteni, hogy az adott teszt objektum ismeretlen tulajdonsága, a tanító halmazból megismert kategóriák közül melyikbe tartozik.

Ezen 3 lépést egy úgynevezett címke (label) alapján hajtjuk végre. Ez egy kitüntetett attribútuma a tanító halmaznak, általában a teszt halmazban ennek az értékét nem ismerjük. A távolságmátrix számítás, és a k legközelebbi elem megtalálása után az az algoritmus feladata,

hogy megtalálja a tanító halmaz címke attribútuma által felvett értékek közül azt, ami a távolságok alapján a lehető legjobban illik a teszthalmaz objektumaira.

Amikor a tanító és a tesztelő halmaz mérete, illetve azok dimenziószáma nagy, általában a futási idő és a memóriaigény legszűkebb keresztmetszet.

3.2.: LÉTEZŐ MEGVALÓSÍTÁSOK

3.2.1 AZ ELSŐ DOKUMENTÁLT CUDA K-NN IMPLEMENTÁCIÓ [1]

Ez az egyik első CUDA k legközelebbi szomszéd algoritmusról szóló publikáció, és sajnos az egyetlen olyan, aminek az írója később megosztotta a forráskódját. Így ezzel a megoldással lehetőségem volt összemérni a következő fejezetben prezentált sajátomat, de erről majd a *6. fejezetben* fogok részletesen írni.

Nagyon jól felismerték az írók, hogy a textúra memória cache-ének alkalmazásával csökkenthetik a nem ***coalesced*** olvasások okozta teljesítmény-kiesést. A cikkben 2D textúrát használnak, ebből olvassák ki a teszt halmaz elemeit, sajnos azonban azt elfelejtik megemlíteni, hogy a textúra maximális mérete Fermi előtt 65535 széles és 32768 magas, Fermi-n a magasság is 65535.

Egy gyors számolást elvégezve, 65535*65535 méretű float értékekkel feltöltött távolságmátrix mérete 16 GB memóriaterületet jelent. Azonban ha valamelyik irányban túllépjük a maximális méretet a kernelünk nem fog lefutni, tehát már egy 65540*5 float-ból álló halmazra se fog lefutni ez a megoldás, pedig ez csak 1,25 MB.

A másik fontos kérdés a cikkel kapcsolatban a memóriaigény kezelése, a cikk bizonyos méréseinél 9600 elemből álló halmazokat használ, maximum 96 attribútummal, ekkor tehát 9600*9600 méretű távolságmátrixra, illetve két 9600*96 méretű tömbre van szüksége, amiben a teszt illetve a tanító halmazok tárolódnak.

Ez körülbelül 400 MB memóriát igényel, ha nem számolunk a textúra méretével, akár azt is feltételezhettük volna, hogy azért pont 9600 elemmel tesztelt, mert több nem fért el a Geforce 8800 GTX globális memóriájában. Később futtat „nagyobb” teszteket, például kettő 38400 méretű halmazra, ennek a memóriaigénye <6 GB, ami az abban az időben elérhető legnagyobb Tesla kártya globális memóriájának méretével egyezett meg.

Ezek csak feltételezések voltak egészen addig, amíg a forráskódot ki nem szerettem volna próbálni a később prezentált programom által könnyen kezelt 50000x50000 méretű távolságmátrixszal, ekkor jött az igazolás a gyanúmra, az „out of memory error”. Tehát ezen cikknek ez az egyik fő tanulsága, hogy a szűk keresztmetszeteket jobban fel kell mérni, például a probléma több kisebb részproblémára bontásával, amivel kiküszöbölhetjük a memória elfogyását.

A következő cikkekben is látszani fog, hogy a k legközelebbi szomszéd 3 fő lépése közül a második legerőforrás-igényesebb a „rendezés”. Erre azért van szükség, mert minden teszt elemhez meg szeretnénk találni a k legközelebbi tanító elemet. Ezt úgy tehetjük meg, hogy a már kiszámított teszt-tanító távolság mátrixban, minden teszt objektumhoz megkeressünk a k legkisebb távolságú tanító objektumot.

Ezt általánosságban a cikkekben két féle módon oldották meg, vagy egy komoly párhuzamos rendező algoritmust (pl.: párhuzamos Radix Sort, Bitonic Sort) futtatnak le egymás

után a ***15. ábrán*** láthatóhoz hasonló távolságmátrix minden sorára. Vagy párhuzamosan futtatnak soros rendezési algoritmusokat az összes sorra egyszerre.

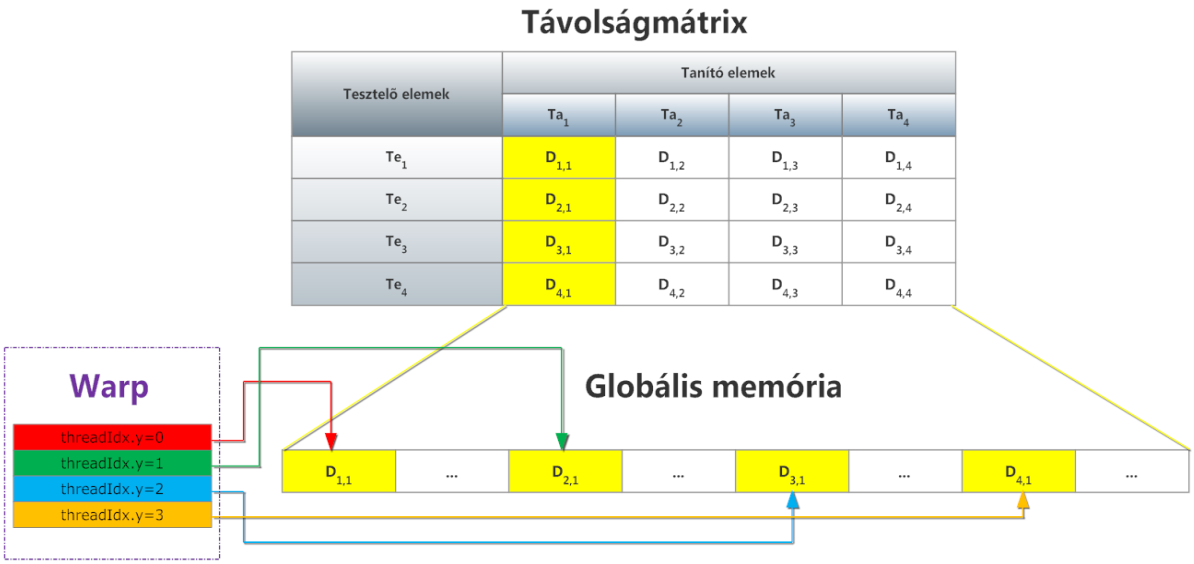
Tesztelő elemek	Tanító elemek			
	Ta_1	Ta_2	Ta_3	Ta_4
Te_1	$D_{1,1}$	$D_{1,2}$	$D_{1,3}$	$D_{1,4}$
Te_2	$D_{2,1}$	$D_{2,2}$	$D_{2,3}$	$D_{2,4}$
Te_3	$D_{3,1}$	$D_{3,2}$	$D_{3,3}$	$D_{3,4}$
Te_4	$D_{4,1}$	$D_{4,2}$	$D_{4,3}$	$D_{4,4}$
Te_5	$D_{5,1}$	$D_{5,2}$	$D_{5,3}$	$D_{5,4}$

15. ÁBRA TÁVOLSÁGMÁTRIX MINTA

Ezen a képen szemléltetve, a párhuzamos rendező algoritmusoknak minden sort egyesével kell rendezniük, de mivel azt részproblémákra bontva párhuzamosan teszik, relatív gyorsan végeznek. Ennek egy alternatívája az, amikor a már jól ismert sorosan működő rendező algoritmusokat egy időben futtatjuk az összes sorra.

A cikk írói soros beszúrásos rendezést futtatnak, az előbb említett módon. Logikátlannak vélem, hogy a textúra memóriát ebben a szakaszban mellőzik. A globális memóriából olvasnak uncoalesced módon minden értéket, miközben a távolságmátrixot textúrából is olvashatnák, valószínűleg ezen megoldással a memóriával spóroltak a szerzők.

Itt hívnám fel az olvasó figyelmét arra a megtévesztő esetre, amikor a szálak folytonos memóriaterületet olvasnak, akkor felmerülhet a kérdés, hogy miért nem tudja ezeket a fordító összevonni egy memóriaolvasássá? A válasz relatív egyszerű, hiszen a coalesced memóriaolvasáshoz a warp-oknak kell folytonos területet olvasnia, nem a szálaknak.



16. ÁBRA UNCOALESCED OLVASÁS SZEMLÉLTETÉSE

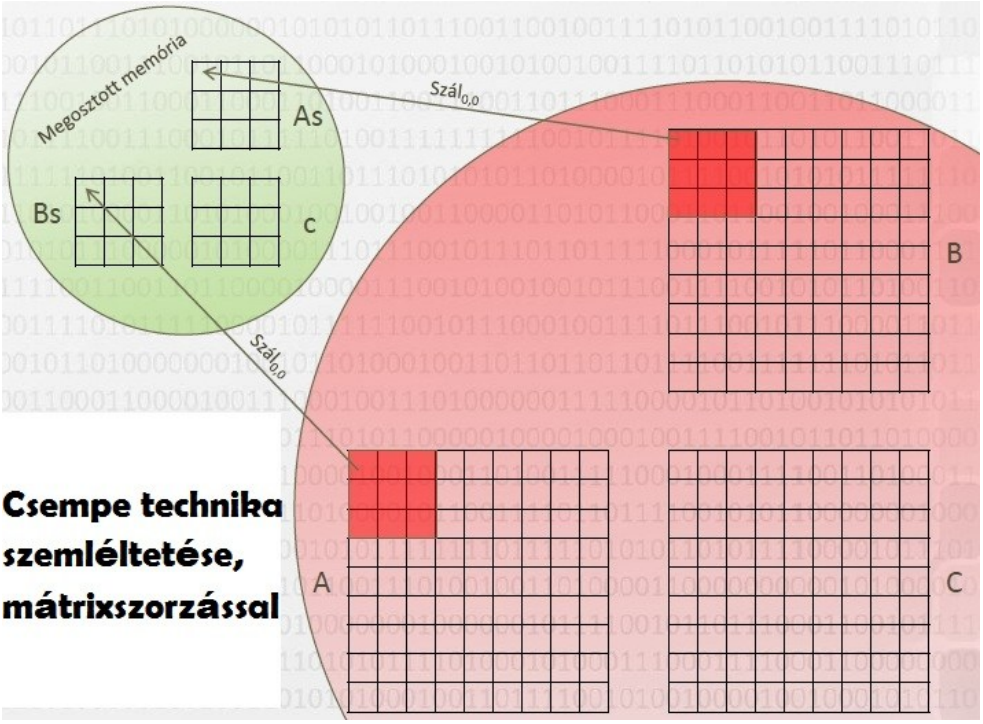
Ezen jelenség jól látható ***16. ábrán***. A távolságmátrix a globális memóriába egy dimenzióban képződik le, így ha egy warp-on belüli szálak egy időben olvassák ki a távolságmátrix sorainak első elemét, akkor mint látható a globális memóriában egymástól távoli lokációkban elhelyezkedő értékeket kell olvasni, tehát uncoalesced olvasásról van szó.

A cikkhez tartozik, hogy a hivatalos weboldaláról letölthető forráskódban, már szerepel egy megosztott memórián alapuló módszer is, aminek megvalósítása komoly kifogásokat hagy maga után, többek között a bank-konfliktusok kezelésének területén. Mivel ez az egyetlen könnyen tesztelhető implementáció, így ezt a *6. fejezetben* részletesen elemzem.

3.2.2. AZ ELSŐ MEGOSZTOTT MEMÓRIÁT HASZNÁLÓ IMPLEMENTÁCIÓ [2]

Az első cikk, ami a megosztott memóriára alapoz, emellett más alternatívát nem használ. Ezt egy úgynevezett csempe technikával valósították meg, aminek az alapelve egy mátrixszorzáson szemléltetve az alábbi ***17. ábrán*** látható.

Első lépésként betöltjük a globális memóriából a bemeneti mátrixok egy-egy azonos méretű rész mátrixát, csempéjét, a megosztott memóriába. Ekkor a megosztott memória a globálishoz viszonyított jóval nagyobb sávszélességét kihasználva elvégezzük a jelen esetben szorzást, majd az eredményt vagy a megosztott memóriában tároljuk, vagy azonnal visszaírjuk a globális memóriában tárolt eredménymátrixba.



17. ÁBRA CSEMPE TECHNIKA SZEMLÉLTETÉSE[15]

A cikkben a megosztott memóriában két 16x16 méretű tömböt és ezzel egyenlő számú, és elrendezésű szálat hoznak létre. Majd ezekbe tölti be a teszt illetve a tanító halmaz 16-16 elemének, első 16 attribútumát. Ezekkel a 16x16-os csempékkel lépkedve dolgozták fel a teszt és tanító halmaz kiválasztott 16 elemének összes attribútumát. Minden távolság kiszámításáért egy szál felel. Korábban már írtam, hogy a megosztott memória alkalmazásánál általában kulcsfontosságú a szinkronizáció.

Ez a cikk megoldásánál sincs másként, minden egyes 16x16-os csempe betöltésénél egy szál csak egy teszt és egy tanító attribútumot tölt be, ebből következik, hogy még mielőtt elkezdenénk a feldolgozást össze kell szinkronizálniuk a szálakat, hiszen könnyen előfordulhat, hogy néhány szál a megosztott memória egy olyan attribútumára hivatkozik, amit egy olyan szálnak kellett volna betölteni, ami egy olyan warp-ban van, ami még nem került futtatásra.

Azonban emellett szükség van még egy szinkronizációra, a csempe feldolgozásának végén, mert meg kell bizonyosodniuk róla, hogy az összes szál elvégezte a feladatát, így be lehet tölteni a következő csempét.

Habár ők nem említik, újra meg kell jegyeznem, hogy a blokkokon belüli szál szinkronizáció egy költséges művelet és ez a megoldás ezt csempénként kétszer végzi el.

Miután az összes csempe feldolgozásával végzett a blokk, az összes $(p_i-q_i)^2$ számítás összegét egy regiszterben tárolják, ezek után már csak gyököt kell vonni belőle az alábbi képletnek megfelelően, és írhatják is vissza az eredményt a globális memóriába.

$$dist(p,q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}$$

Nagy hiányossága a cikknek, hogy a coalesced memóriaolvasások jelentősége, még említőlegesen sem szerepel benne, így feltételezhető, hogy a tervezés során ezt a nagyon fontos elvet nem tartották szem előtt.

Ezzel ellentétben a megosztott memória bankkonfliktusainak kezelésére állításuk szerint odafigyeltek, hogy pontosan hogyan azt nem részletezik.

A mérésekből itt is kiderül, hogy nem gondoltak a GPU memóriájának mérete szűk keresztmetszet lehet. A maximális távolságmátrix, amit kiszámolnak 30296x2265 méretű, ez a GPU memóriájában 262 MB helyet foglal, emellett esetleges darabolásról nem említenek egy szót sem.

A legkisebb k keresésnél a távolságmátrix minden sorára egy párhuzamos Radix rendezést futtatnak, majd a teljes rendezett sorból csak az első k elemet használják fel.

Végezetül a k-NN algoritmus utolsó lépését, a megfelelő címke kiválasztását, már a CPU-n végzik, mert „ezt nehéz optimalizálni GPU-n”, illetve a méréseik szerint „nem vesz több igénybe több időt, mint 20 ms”.

3.2.3 A LEGVERSENYKÉPESEBB MEGVALÓSÍTÁS [3]

Nem véletlenül hagytam ezt a cikket a végére, hiszen az előző fejezetben bemutatott megoldások közül ez vonultatja fel a legtöbbet.

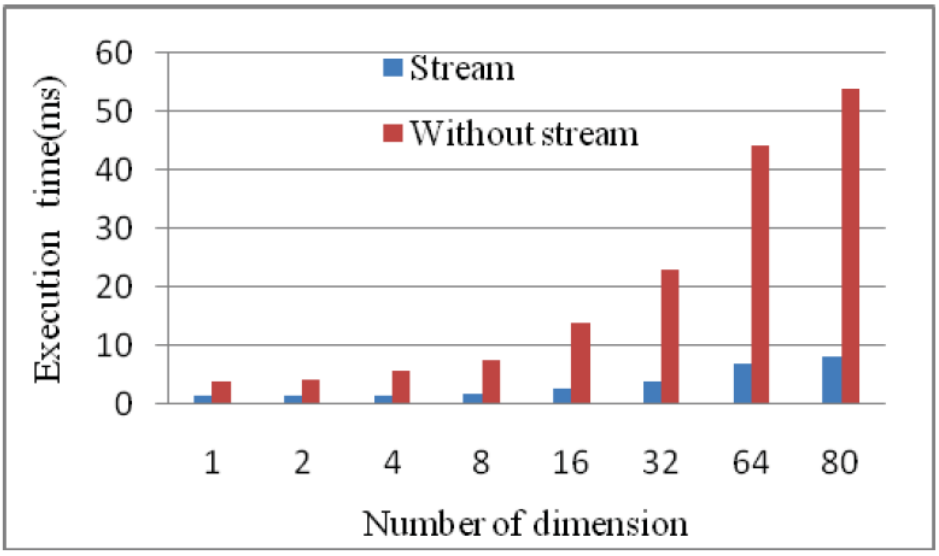
Azonban ez se nevezhető teljes értékű megoldásnak, a távolságmátrix számítást ugyan azzal a szinkronizációktól hemzsegő megosztott memóriás módszerrel számolja, mint az előző cikk implementációja, de kiegészítették azt egy komoly változtatással, a memóriaolvasások coalesced módon történnek.

A rendezésre egy nagyon egyedi megoldást használnak, a blokkok a rész eredménymátrixot is a megosztott memóriában tárolják, az utolsó attribútum csempe kiszámolása után beírják a megosztott memóriába a végleges eredményeket, majd

szinkronizálják a szálakat. Ekkor az összes szál látja, az eredményeket, és állításuk szerint ki tudja keresni a nagyság szerint a saját pozícióját, illetve el tudja dönteni, hogy benne van-e a legkisebb k elemben. Ezek után már csak a legkisebb k elemet kell visszaírni a globális memóriába, és ott egy meg nem adott, valószínűleg soros kereső algoritmus keresi ki a globálisan k legkisebb távolságú tanító elemet minden teszt elemhez.

Az első és legfontosabb probléma ezzel az, hogy habár felhívják külön az olvasó figyelmét, hogy milyen sokat használják a megosztott memóriát és ez milyen hatékony megoldás, de egyszer sem tesznek említést arról, hogy a bankkonfliktusok kivédése érdekében mit tettek.

Ez az első cikk, ami felismeri, hogy előnyt lehet szerezni abból, ha a GPU-ra másolandó adathalmazokat felbontjuk több részre és a feldolgozást átlapoljuk a másolással. Sajnos azonban, a ***page-locked*** memória, mint fogalom abszolút nem szerepel benne, felismerik, hogy a kernel futásának jelentős részét a CPU<->GPU másolások teszik ki, ezért stream-ek alkalmazásához folyamodnak. Eddig nem írtam arról, hogy ha aszinkron másolást szeretnénk végrehajtani, akár egy stream keretein belül, akkor az csak page-locked memóriával fog működni.



18. ÁBRA HIBÁS KÖVETKEZTETÉSŰ STREAM ÉS NEM STREAM ÁBRA[3]

A ***18. ábrán*** látható a mérésük eredménye, de véleményem szerint, önmagában stream-ek alkalmazásával ilyen mértékű gyorsulást nem lehet elérni, kivéve, ha több kernelt futtatunk konkurensen, de akkoriban erre még nem volt lehetőség. Itt a háttérben valószínűleg a page-locked memóriafoglalás volt.

A megfelelő label megtalálását a publikáció írói is a CPU-n számolták.

3.2.3 R+GPU, GPGPU PLUGIN AZ R STATISZTIKAI PROGRAMNYELVHEZ

Az R+GPU az egyetlen jelenleg piacon lévő adatbányászati alkalmazáshoz készített GPU plugin. Ez az R-hez készült, ami egy nyílt forráskódú adatbányász és statisztikai programnyelv, meglehetősen összetett, kezelése komoly szakértelmet kíván. Sajnos emiatt nem sikerült a plugin-t telepítenem, de szerencsére a forráskódját látva megállapíthattam, hogy miről maradtam le.

Azt hiszem talán a legfontosabb, legszembetűnőbb negatívum, hogy ezen plugin fejlesztői se vették figyelembe, hogy a globális memória mérete elég komoly szűk keresztmetszet, a weboldalukon ennek megfelelően 4500x4500 a maximális távolságmátrix, amit számolnak, a forráskód is ezt igazolja. A funkciói között szerepel a távolságmátrix számítás, 5 távolságfüggvénnyel, többek között az euklideszi és Manhattan távolsággal. Az implementációt megvizsgálva, egy meglehetősen kezdetleges megoldást lehet megfigyelni, úgy néz ki a készítő, még az előbb említett 3 cikket és azok eredményeit sem ismerte.

A teszt és a tanító halmaz attribútumait a globális memóriából olvassa uncoalesced módon, a megosztott memóriában csupán 32 részeredményt tárol, azért pont ennyit, mert egy blokkban 32 szálat, azaz egy warpot hozott létre. Ezzel teljesen ellehetetlenítette az egyébként sem coalesced olvasások késleltetésének elfedését. Sőt, ezt még felül tudja múlni azzal, hogy a megosztott memória műveletek után általában összeszinkronizálja a szálakat, aminek semmi értelme, hiszen csak egyetlen warp-ja van, amiben a szálak egyébként is párhuzamosan futnak, tehát nincsenek további warp-ok a blokkban, amik esetleg még nem futottak le.

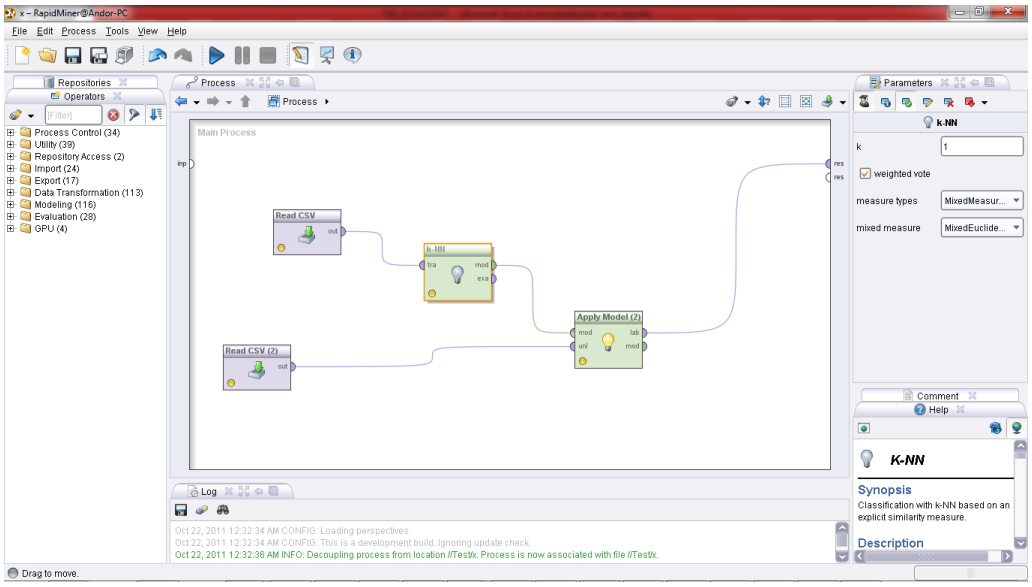
4. A PLUGIN TERVEZÉSE:

A tervezés megkezdése előtt röviden bemutatom a kiegészítendő szoftvert a RapidMinert, és azokat a szempontokat, amik miatt erre esett a választás.

4.1 A RAPIDMINER RÖVID BEMUTATÁSA

A RapidMiner egy nyílt forráskódú szoftver, ami adatbányászati funkciók széles választékát nyújtja a felhasználók számára. Tervezése során két szempontot tartottak szem előtt a fejlesztői, elsőként, hogy minél egyszerűbben használható legyen, ezt a ***19. ábrán*** is megfigyelhető, jól átlátható, ezt felhasználóbarát grafikus felülettel érték el. A felhasználónak nem kell összetett parancssoros felület, illetve az ahhoz tartozó scriptnyelvek, használatát megtanulnia.

A másik fontos tervezési szempontja az volt, hogy minél egyszerűbben lehessen bővítményeket, plugin-okat fejleszteni hozzá, azok telepítése a lehető legegyszerűbb legyen.



19. ÁBRA KÉPERNYŐKÉP A RAPIDMINER SZOFTVERRŐL

Ezen két szempont miatt esett a választás a RapidMiner-re. Mivel ez egy ingyenes, könnyen kezelhető szoftver, így egy esetlegesen más adatbányászati eszközhöz szokott felhasználó is könnyen válthat erre, hogy a plugin által nyújtott sebességnövekedésből profitáljon.

Az sem volt mellékes a választásnál, hogy a KDnuggets adatbányászati közösségi oldal 2011 májusi felmérése szerint, az olvasók körében a RapidMiner volt a leggyakrabban használt szoftver. [16]

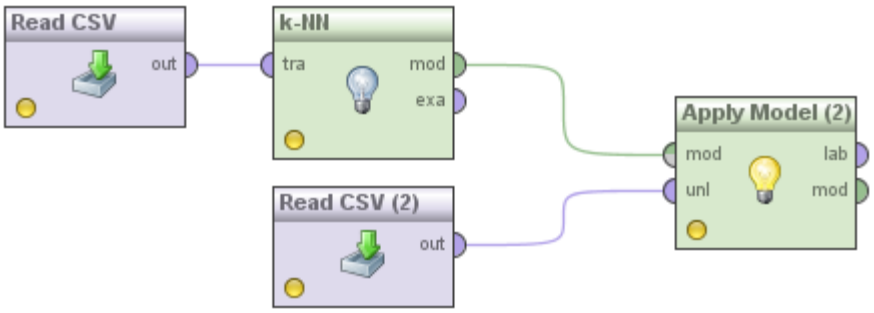
4.2 A RAPIDMINER K-NN FUNKCIÓJÁNAK ELEMZÉSE

A RapidMiner funkcióit operátornak hívják, ezeket a felhasználók „dobozok" formájában érhetik el, ilyenekből tetszőleges számút helyezhetnek a grafikus felületre. Ezen operátoroknak, mint a ***20. ábrán*** látható kimenetei és bemenetei vannak, amiket a felhasználó tetszőlegesen összeköthet, ezeken az összeköttetéseken keresztül adja át az egyik operátor az eredmény objektumának referenciáját a másik operátornak.

4.2.1 A K-NN OPERÁTOR HASZNÁLATA A FELHASZNÁLÓ SZEMPONTJÁBÓL

Ha a felhasználó egy k-NN keresést szeretne végrehajtani, akkor az input fájlokat beolvasó operátorokon kívül (***20. ábrán*** „Read CSV") további kettőre lesz szüksége. Egyik ezek közül a „k-NN", aminek a bemenetére kell kötni a tanító halmazt, ami alapján létrehoz egy modellt, ami alatt azt kell érteni, hogy a tanító halmazt átalakítja olyan formába, hogy a k-NN számolása könnyen elvégezhető legyen rajta.

A modellbe belecsomagolja a k-NN operátor beállításait, tehát, hogy milyen távolság függvénnyel szeretne a felhasználó számolni, szeretne-e súlyozást és hogy mekkora a k értéke. Ezt a modellt adja át az „Apply Model"-nek. Ezen operátor a modellben kapott tanító halmazon, és a közvetlenül a bemenetére kötött teszt halmazon, a modellbe csomagolt beállításokat figyelembe véve elvégzi a k-NN számítást, majd az eredményt a „lab" kimenetre juttatja.



20. ÁBRA A K-NN OPERÁTOR HASZNÁLATA A GRAFIKUS FELÜLETEN

4.2.2 AZ OPERÁTORRAL SZEMBENI ADATBÁNYÁSZATI KÖVETELMÉNYEK

A legfontosabb követelmény az, hogy a GPU-s operátor képes legyen mindenre, amire a CPU-s társa, és ezek azzal teljesen megegyező módon vehessék igénybe a felhasználók. A korábban említett cikkek csak numerikus attribútumok kezelésére voltak alkalmasak, ez komoly használat esetén nem elegendő, így szükség van a nominális attribútumokkal való számolás támogatására.

Az adatbányászatban, komoly gondot jelenthetnek a hiányzó (NaN) „értékek”, így ezek kezelésére is komoly szükség van.

A k-NN operátor alapvetően két féle modellt hozhat létre attól függően, hogy tanító halmaz kitüntetett label attribútuma nominális vagy numerikus típusú. Nominális típusnál osztályozást kell végrehajtani, tehát a meglévő label értékekhez kell hozzárendelni (osztályozni) a teszt halmaz elemeit. Numerikus típusnál ezt nem lehet megtenni így ekkor regresszióval közelíteni kell, így az eredmény egy lebegő pontos érték lesz a teszt halmaz minden eleméhez.

CPU-n lehetőség van súlyozni az eredményt, ez nem maradhat ki a GPU verzióból sem.

A jelenlegi megoldás a távolságfüggvények széles választékát vonultatják fel, habár valószínűleg az euklideszi távolság a legnépszerűbb, lehetőséget kell adni a választásra.

Illetve végül talán a legfontosabb követelmény a korábbi cikkeken jóval túlmutató skálázhatóság, aminek kielégítése munkám talán legnagyobb eredménye. Értve ez alatt, azt, hogy mivel ezen GPU program nem egy specifikus területre és alkalmazásra készül, ahol jól be lehet határolni azt, hogy a bemenet minimális, illetve maximális mérete mely tartományokban változhat, így a lehetőségekhez képest tisztességesen helyt kell állni még akkor is, ha a bemenet mérete több nagyságrendet változik.

Változók, amiket a tervezésnél figyelembe kell venni:

- Tanító halmaz elemeinek és attribútumainak száma
- Teszt halmaz elemeinek és attribútumainak száma
- k értéke

4.2.3 AZ OPERÁTORRAL SZEMBENI GPGPU KÖVETELMÉNYEK, AZ EDDIGI MEGOLDÁSOKBÓL OKULVA

Ami az előző cikkek alapján kétségtelen, hogy az előző pontban felsorolt bemeneti paraméterek akár több nagyságrendű változásának hatékony követésére egyetlen implementált kernel, teljesítmény szempontjából csak lokális optimumot tud adni. Erre legegyszerűbb példa a Textúra memória, amit csak akkor lehet alkalmazni, ha a bemenet belefér vagy azt „bele lehet darabolni”, ennek komolyabb részleteire később térek ki.

Kétségtelen követelmény, hogy lehetőség szerint több GPU architektúrára legyen optimalizálva. Továbbá GPU specifikus követelmény, az hogy az előző cikkekből tanulva, ha globális memória olvasásra kerül sor, azt mindenképp ***coalesced*** módon kell tenni, a megosztott memóriánál pedig figyelni kell a bankkonfliktusokra.

Azt az időt, amíg a RapidMiner-ből kiolvassuk és másoljuk az adatokat a GPU-ra el kell fedni a feldolgozásokkal stream-ek , vagy aszinkron működés segítségével, ennek érdekében a korábban említett page-locked memóriát is fel kell használni, de lehetőség szerint korlátozott mértékben, korábban említett okok miatt.

A publikációkban olvasható megoldásoknál látható volt, hogy ha a problémát nem bontjuk fel részproblémákra, akkor a párhuzamos futtatásnak köszönhetően már relatív kis bemenetek esetén is elfogy a GPU memóriája, így a plugin-nak ezt is hatékonyan meg kell tudnia oldani.

4.3 A RAPIDMINER ÉS A CUDA ÖSSZEKÖTÉSE, KOMMUNIKÁCIÓJA

A plugin-nal kapcsolatos általános követelmények meghatározása után röviden rátérek arra, hogy azt miként illesztettem a RapidMiner-hez.

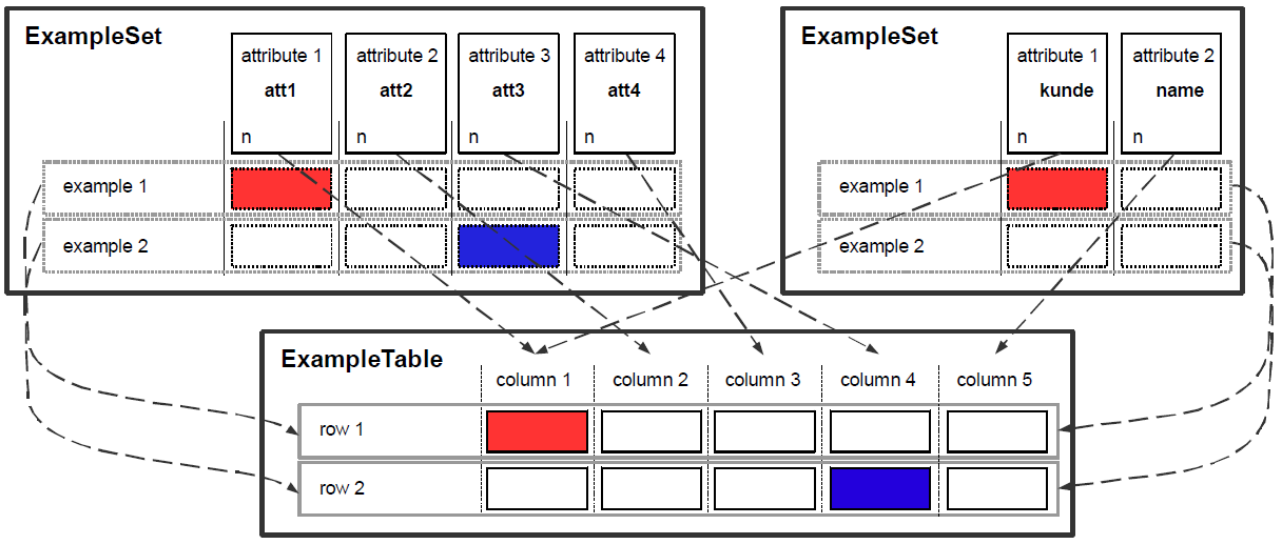
4.3.1 A JCUDA RÖVID BEMUTATÁSA

A CUDA kód Java-ból történő futtatásához a JCUDA[*5*] nevű illesztést használtam, ez a Java Native Interface segítségével elérhetővé teszi a CUDA alapvetően hoszton futó, vagy onnan indítható függvényeit, mivel a Java referenciák nem elég rugalmasak, így definiál két pointer osztályt, egyet a CPU, egyet pedig a GPU memóriához.

Alapvetően a korábban már említett Driver API-t használja, ezen keresztül van lehetőségünk a külön NVCC-vel lefordított CUDA modulokat betölteni, és futtatni.

4.3.2 ADATTÁROLÁS A RAPIDMINER-BEN

Az adattárolás alaposztálya a ***21. ábrán*** látható *ExampleTable*, ebben az összes érték nyers adat formájában dupla pontosságú lebegőpontos alakban tárolódik. Memóriatakarékosság érdekében ezek a nyers adatok nem duplikálódnak, csupán *ExampleSet* nevű nézetek jöhetnek hozzájuk létre. Ezek az *ExampleTable* sorainak és oszlopainak tetszőleges részhalmazát, tetszőleges sorrendben tartalmazhatják. Ez jól megfigyelhető az ábrán is.

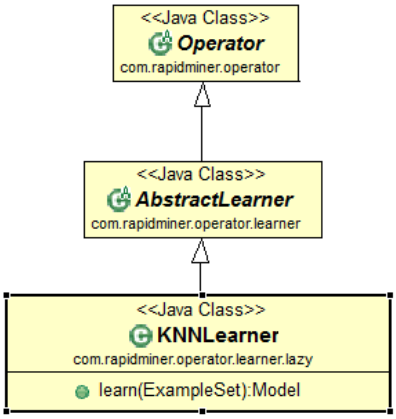


21. ÁBRA ADATTÁROLÁS A RAPIDMINER-BEN[17]

4.3.3 CPU K-NN OPERÁTOR FELÉPÍTÉSE, GPU K-NN ILLESZTÉSÉNEK HELYE

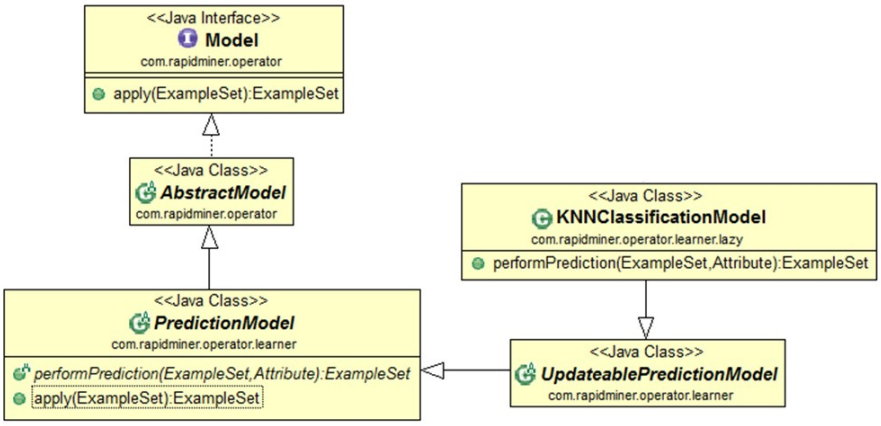
A kitűzött cél ezen a területen az volt, hogy a felhasználó ugyanúgy tudja használni a GPU-n futó operátort, mint azt korábban megszokta, ehhez vázlatosan meg kell ismerni a CPU-s megfelelőjét.

A ***22. ábrán*** látható a *KNNLearner*, ami a k-NN operátort megvalósítja ennek a legfontosabb függvénye az *learn*(), ami a bemenetre kapott tanító *ExampleSet*-ből egy modellt hoz létre, és ezt a kimenetre adja.



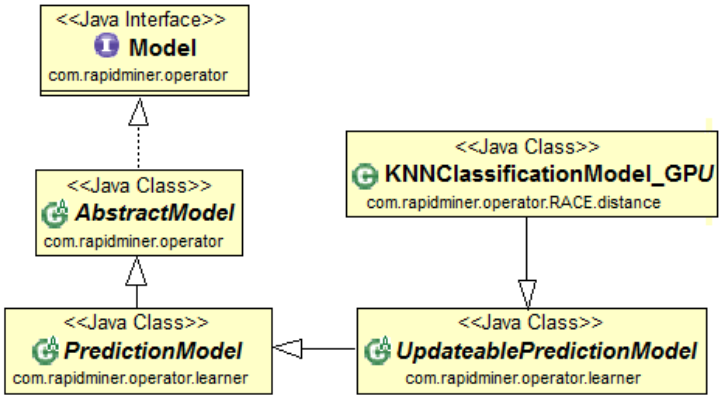
22. ÁBRA CPU K-NN OPERÁTOR

A ***23. ábrán*** látható, hogy a ***Model*** interfész *apply()* függvényét a ***PredictionModel*** definiálja, ami meghívja a *performPrediction()* absztrakt függvényt, amit klasszifikációs esetben a ***KNNClassificationModel*** definiál.



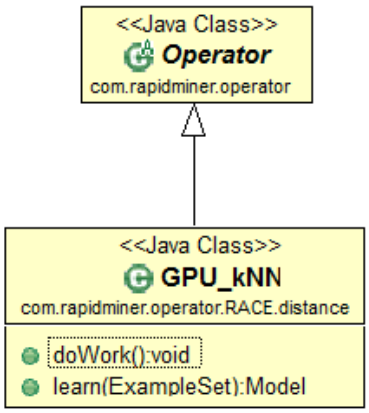
23. ÁBRA KNNCLASSIFICATIONMODEL OSZTÁLYDIAGRAMON

Ezen *apply()* függvényt a korábban látott „Apply Model" operátor hívja meg, és adja át neki paraméterként a bemenetére kapott tesztelő *ExampleSet*-et. Ennek ismeretében rájöttem, hogy elegendő egy új a ***24. ábrán*** látható GPU specifikus modell objektumot létrehoznom, ami szintén implementálja a *Model* interfészt, így rendelkezik *apply()* függvénnyel, ami a programban a GPU-n történő feldolgozást indítja.



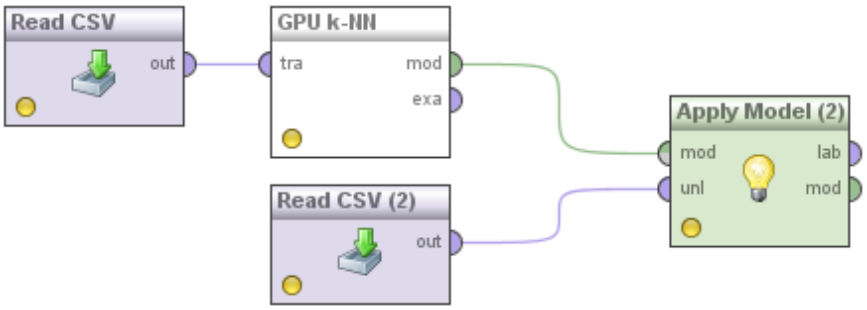
24. ÁBRA GPU SPECIFIKUS MODEL OBJEKTUM

Ezen objektum birtokában már csak egy ezt létrehozó és „Apply Model”-nek átadó, ***25. ábrán*** látható „GPU k-NN” operátort kellett létrehoznom.



25. ÁBRA GPU K-NN OPERÁTOR

A ***26. ábrán*** láthatjuk, hogy a felhasználó szinte semmi különbséget nem fog látni a kész GPU operátor és a CPU-s megfelelő között. Alkalmazásuk pedig teljesen egyező, azonban a következő pontban bemutatott belső működés tervezésekor az volt a célom, hogy azoknak a felhasználóknak, akik egyszer már kipróbálták ezt és utána vissza kell térniük a CPU-s megfelelőhöz komoly hiányérzete legyen.



26. ÁBRA A KÉSZ GPU K-NN A GRAFIKUS FELÜLETEN

5. JCUDA IMPLEMENTÁCIÓ ELKÉSZÍTÉSE

Ebben a fejezetben két távolságfüggvény GPU implementációját, és az azokat körülvevő keretrendszert fogom bemutatni. Egyik ezek közül a korábbi implementációkban is szereplő euklideszi távolság, a másik pedig annak egy a RapidMiner-ben szereplő kiegészítése a „kevert” euklideszi távolság (MixedEuclideanDistance). Utóbbi jelentősége abban rejlik, hogy képes nominális attribútumok kezelésére is, míg „klasszikus” megfelelője csupán numerikusakkal képes számolni.

A nominális attribútumok azért igényelnek külön kezelést, mert habár ezen attribútumok is lebegőpontos számként tárolódnak nincs értelme ezek között is a numerikus értékeknél megszokott távolságot számítani, mert például az „alma” és a „banán” illetve az ezeket reprezentáló számok között nem tudjuk a pontos távolságot meghatározni. De ennek ellenére valahogyan ki kell fejeznünk, hogy különböznek egymástól, ezt a RapidMiner úgy teszi meg, hogy ha két nominális érték nem egyezik meg, akkor növeli eggyel az össztávolságot.

Ez egy CPU kódban meglehetősen egyszerű módosítás és a teljesítményre való hatása is könnyen megbecsülhető, azonban, mint a következő fejezetben látszani fog a GPU-n teljesen más a helyzet.

Ahogy korábban említettem az előző cikkekből azon következtetést vontam le, hogy egy implementációval csupán egy bizonyos bemenet tartományban lehet optimális teljesítményt elérni, ezért célom volt megtalálni azon módszereket és implementációs technikákat, amikkel a korábbi megvalósítások optimalizált verzióit ötvözve még szélesebb tartományban érhető el optimális teljesítmény.

Ezen implementációs technikák két területen különböznek, abban hogy a GPU melyik memóriatípusát és hogyan használják. Ezek a technikák a következők:

- *Olvasás 1 dimenziós textúrából*
- *Olvasás 2 dimenziós textúrából*
- *Olvasás coalesced globális memóriából megosztott memóriába „Mindent betölt” stratégiával*
- *Olvasás coalesced globális memóriából megosztott memóriába „Részletet tölt be” stratégiával*
- *Olvasás 1 dimenziós textúrából megosztott memóriába „Részletet tölt be” stratégiával*

Ezek közül utóbbi részletes működését fogom elemezni, majd rávilágítok azokra a tényezőkre, amik miatt szükség van a többi megvalósításra. Ennek a működése a legösszetettebb, így színes ábrákkal mutatom be a főbb lépéseket, felhívom a figyelmet arra, hogy konzisztens színkonvenciót használtam, így a folyamat első ábrájától az utolsóig végigkövethető, hogy egy adat miként hat, vagy hathat a végeredményre.

5.1 ADATHALMAZOK TÁROLÁS A GPU-N

Korábban írtam a ***coalesced*** olvasások jelentőségéről, illetve arról, hogy ennek elérése érdekében az „egyszerre” kiolvasandó adatokat tanácsos egymás mellé helyezni. Tudjuk, hogy az euklideszi távolság mindig az egyik halmaz i. attribútumát a másik halmaz i. attribútumával hasonlítja össze, hogy ezt elősegítsem a ***27. ábrán*** látható Teszt halmaz logikai mátrixa helyett a transzponáltját másoltam a GPU-ra. Ennek köszönhetően látható, hogy az azonos színnel jelölt attribútumok az egy dimenziós memóriaterületen egymás mellé kerültek, ez azért fontos, mert ugyebár a *warp*-oknak kell folytonos memóriaterületet olvasnia a ***coalesced*** olvasás érdekében, ez ilyen elrendezéssel, mint később látható könnyen megvalósítható.

27. ÁBRA A TESZT HALMAZ TÁROLÁSA A GPU MEMÓRIÁJÁBAN

A ***27. ábrán*** az is látszik, hogy a teszt halmaz a globális és a textúra memóriában is tárolódhat, ezek teljesítménykülönbségéről szintén később értekezek.

A ***28. ábrán*** látszik, hogy a tanító halmazt is transzponált formában tárolom, hogy az azonos attribútumok egymás mellé kerüljenek. Emellett látható, hogy erről sokkal több mindent le kell tárolni. Fontos elem a „LABEL", vagyis a címke attribútum, hiszen a teszt halmaz elemeit (sorait), a címke által felvett értékek egyikében szeretnénk besorolni.

Tehát a képen látható egyszerű esetben L1-be vagy L2-be. Ezt már a folyamat elején bemásolom a GPU memóriájába, teszem ezt azért, mert a tanító halmaz a CPU memóriában page-locked módon van lefoglalva, a CPU->GPU másolási idő lecsökkentése érdekében, így miután a képen látható három adathalmazt a GPU-ra másoltam a tanító halmaz CPU memóriája felszabadíthatóvá válik.

28. ÁBRA A TANÍTÓ HALMAZ TÁROLÁSA A GPU MEMÓRIÁJÁBAN

A címkék mellett a másik nagyon fontos adattömb arról tartalmaz információt, hogy az adott attribútum numerikus vagy nominális típusú. Mivel ezen adatokra nagyon gyakran van szükség így a konstans memóriába helyeztem őket, hiszen nem változnak és pontosan megfelel az optimális elérési mintának, amit a *2. fejezetben* ismertettem a konstans memóriáról.

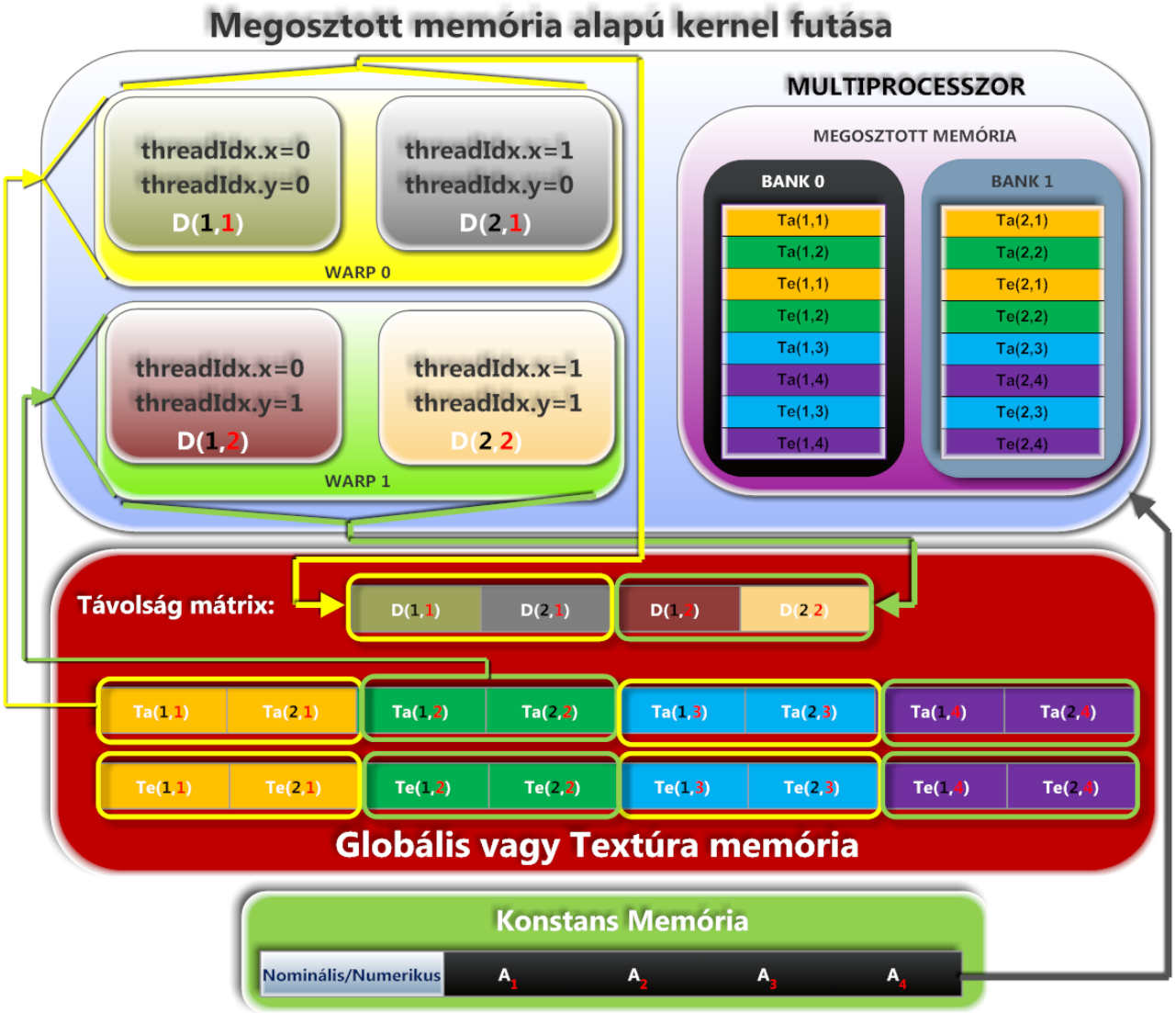
Ez egy nagyon hatékony döntés volt, hiszen a warp-ok az euklideszi távolság miatt általában egyszerre ugyan azt az attribútumot olvassák, csak maximum más elemhez tartozót, így a konstans memóriának ugyanarra a szavára fognak hivatkozni, ekkor broadcastolás léphet fel.

Azonban mivel ennek 64 KB a mérete, ebben csak 16384 attribútumról tárolhatunk adatokat, ha a felhasználó ennél többet szeretne feldolgozni, például idősorok esetén, akkor a konstans memória helyett egy dimenziós textúrából töltődnek be ezek az értékek. Sajnos az nem olyan gyors, mint a konstans memória, de a robusztusság érdekében ezt is biztosítani kell.

5.2.1 A MEGOSZTOTT MEMÓRIÁBA VALÓ COALESCED BANK-KONFLIKTUS MENTES BETÖLTÉS

A ***29. ábra*** értelmezését globális vagy textúra memóriában található tanító illetve teszt halmazoktól érdemes kezdeni. Ezekre sárga illetve zöld téglalapokat helyeztem, amik arra utalnak, hogy a kép tetején látható multiprocesszor kettő futó *warp*-ja közül melyik tölti be őket.

Itt komoly egyszerűsítés, hogy csak kettő szál van egy *warp*-ban, illetve a megosztott memória csak két bankból áll, de mivel ez a két szám megegyezik, így a példa valószerű. Ugyebár *Fermi*-n 32 szál és 32 bank van, előtte pedig 32 szál, illetve 16 bank van, de a konkurens memória műveletek a warp-ok első és második felében nem okozhatnak egymás között bank-konfliktust.



29. ÁBRA ÖSSZETETT ÁBRA A COALESCED BANK-KONFLIKTUS MENTES MŰKÖDÉSRŐL

Látható, hogy minden szál egy elemet olvas ki a memóriából, és a *warp*-ok egységes területet olvasnak, tehát az olvasás ***coalesced*** módon történik. A bank-konfliktusokra a warp-okon belül kell figyelni, több *warp* között nem léphet fel. Így azt kellett garantálnom, hogy egy

warp-on belül több szál ne hivatkozzon ugyanarra a bankra egy időben, kivétel ez alól, ha a bankon belül ugyanarra a szóra hivatkoznak hiszen, akkor az az érték *broadcastolódik* és nem okoz bank-konfliktust. Ezt úgy tettem meg, hogy a szálak számát a képen látható módon úgy határoztam meg, hogy minél könnyebben lehessen egy a megosztott memóriában tárolt 2 dimenziós tömböt úgy indexelni velük, hogy azok a tömb olyan elemeire mutassanak, amik különböző bankokban találhatóak. Általában a megosztott memóriában sorfolytonosan tárolt tömbök szomszédos szavai különböző bankokban találhatóak, így ha egy tömb „sorait" címezzük meg a szálak azonosítóival, akkor az kielégíti ezen követelményt.

Látható a ***29. ábra*** megosztott memóriájában a globális memóriából ***coalesced*** módon egyszerre kiolvasott memóriaszavak különböző bankokba kerültek. Négy szálat hoztam létre, egy 2x2 méterű távolságmátrixot kell előállítani, tehát minden távolság számításáért egy szál felelős. Ezek a blokkon belüli pozíciójuknak megfelelő elemét számolják ki a mátrixnak. Majd ezt szintén transzponáltan írják vissza a globális memóriába, hiszen azt még a következő lépésnek is ki kell olvasnia.

A szálak betöltéskor először a 2x2es szálszerkezetnek megfelelően 2x2 méretű tanító halmaz részletet töltenek be, majd 2x2 méretűt a teszt halmazból. A valós implementációnál 16x16 szálat használtam, és egyszerre 16x16 elemet töltöttem be egy logikai lépéssel, ami *warp*-ok szintjén 2x16 elemet jelent, mivel azonban ebből 16osával hajtódnak végre az írási és olvasási műveletek és 16 bank van, így ha a warp-ok a megosztott memóriában folytonos területet írnak, akkor nem léphet fel konfliktus.

Ferminél szintén nem, habár ott már a szál első és második fele okozhat problémát, de 32 bank van tehát ez a folytonos területes módszer ott is működik. Természetesen a nagy méterű működéshez egy multiprocesszor egy blokkja nem elég, így a korábban bemutatott csempe módszerrel skálázom a távolságmátrixot a teljes GPU-ra.

A korábbi cikkekhez képest munkám egy új megosztott memóriás módszert prezentál és azt kombináltan használja a meglévő implementációval.

5.2.2 KÜLÖNBÖZŐ CACHE-ELÉSI STRATÉGIÁK

Mivel a megosztott memória egy olyan cache-nek tekinthető, aminek a menedzseléséért a programozó felelős, így érdekesnek vélem, hogy a cikkekben nem vizsgálták meg alternatív cache stratégiák lehetőségét. Habár lehetőség komoly stratégiák megvalósítására sajnos nincs, mivel ezek komoly erőforrásigénnyel járhatnak, de jelen esetben kézenfekvő megoldás egy úgynevezett „Mindent betölt" módszer implementálása. Ez alatt azt kell érteni, ha a 16x16 méretű távolságmátrix csempe összes attribútuma befér a megosztott memóriába, akkor a kernel nem egy iteráló 16x16 attribútumot betöltő, majd azonnal feldolgozó futásba kezd, mint a „Részletet tölt be" módszernél. Ehelyett a kernel csupán két lépésből áll, egy betöltésből, amikor a 16 tesz és 16 tanító halmaz elem összes attribútumát betölti, majd egyszer összeszinkronizálja a szálakat, és utána akadálymentesen végezheti a feldolgozást.

Ezzel a módszerrel azért lehet teljesítményelőnyt szerezni, mert a „Részletet tölt be" kernel jelentősen utasításigényes, a *2.3.5 pontban* bemutatott kernelt limitáló tényezők közül, a maximálisan futtatható utasításszám által limitált kategóriába tartozik. Tehát annyira leterheli a GPU aritmetikáját, hogy nem tud elegendő memória olvasást kezdeményezni, hogy a GPU buszát kellőképpen kihasználja. Habár a „Mindent betölt" módszert is ez limitálja, de jelentősen kevesebb műveletet kell végrehajtania, így jobb teljesítményt tud elérni. Persze ezen módszer használatához dinamikusan kell foglalnunk a megosztott memóriát, mivel mindig 16x16 méretű csempét számolunk, így a teljesítményt befolyásoló tényező az attribútumok száma. Ennek

növekedésének hatására, a blokkok megosztott memória szükséglete is arányosan nő, ez a korábban bemutatott példának megfelelően hatással van a kernel ***occupancy***-jére. Minél nagyobb az attribútumok száma annál kevesebb blokk futhat egy multiprocesszoron, így csökken az ***occupancy***. Egy bizonyos szint alatt, pedig már a *2.3.5 pontban* bemutatott másik tényező a késleltetés játsza a nagyobb szerepet, tehát nem lesz elég warp, amit be lehetne ütemezni, hogy elfedjük velük a globális memória késleltetését. Emiatt szükség van mindkét implementációra, az egyik „kisebb" attribútum számokra optimális, a másik a nagyokra.

5.2.3 BANK-KONFLIKTUS MENTES FELDOLGOZÁS A MEGOSZTOTT MEMÓRIÁBÓL

Láthattuk, hogy a megosztott memóriába való betöltés nem jár konfliktussal, azonban a feldolgozásnál ez már nem ilyen triviális, így a ***30. ábrán*** bemutatom, hogy az egyik *warp* szálai hogyan kooperálnak egymással.

30. ÁBRA BANK-KONFLIKTUS VIZSGÁLAT

Az x=0 y=0 azonosítójú szál feladata a távolságmátrix ***(1,1)*** eredményének kiszámítása, ehhez a tanító és teszt halmaz első elemét kell összehasonlítania, a rajzon: ***Ta(1,x)*** és ***Te(1,x).*** Az ábra két oldalán a táblázatokban a szálak által kiolvasandó adatok szerepelnek, illetve az, hogy melyik bankban hányadik memóriaszóban találhatóak. A két táblázat sorai párhuzamosan hajtódnak végre. Ha a két szál más bankhoz fér hozzá, akkor biztosan nem lép fel konfliktus, ezt zölddel jelöltem, pirossal azt, ha azonos bankhoz férnek hozzá, mert ilyenkor felléphetne probléma, ha nem ugyanahhoz a szóhoz férnének hozzá, mivel ilyen nem lép fel, így mindegyik olvasás konfliktusmentes volt.

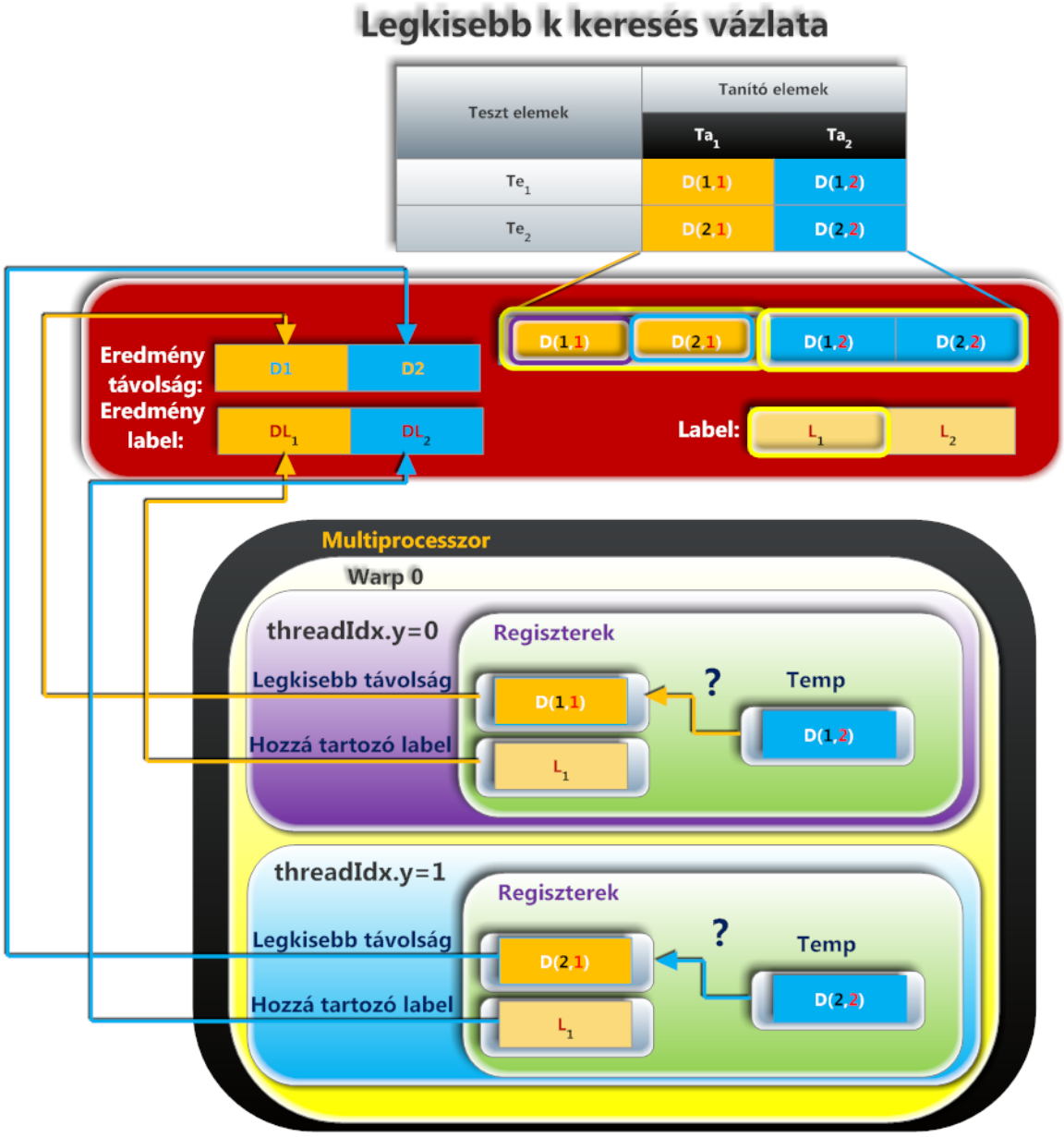
5.3 LEGKISEBB K ÉRTÉK KERESÉSE

Ez a második legerőforrás-igényesebb folyamat, erre egy teljesen egyedi módszert találtam ki, alapvetően a regiszterek nagy sebességére, illetve az utasítás szintű

párhuzamosságra alapozva, a módszerem egy multiprocesszor összes rendelkezésre álló regiszterét próbálja használni.

A cél a ***31. ábra*** tetején látható (Teszt elemszám)x(Tanító elemszám) méretű távolságmátrix, minden sorában megtalálni a k legkisebb távolságot, tehát az adott teszt elemhez „legközelebb" álló tanító elemeket.

Ehhez le kell tárolni a legközelebbi elemek távolságát, illetve azok címkéit, amit még indításkor töltöttünk a globális memóriába, és mint látható az ábrán ez azóta is ott van. Ha a blokk méretét 256-nak vesszük, jelen esetben függőleges orientációval, akkor minden szálnak a távolságmátrix egy sorát kell végigolvasnia, ami persze transzponálva van tárolva, a coalesced olvasás miatt.



31. ÁBRA K LEGKISEBB ELEM KERNEL VÁZLATA

Mivel minden szál egy sort dolgoz fel, nem egy párhuzamos algoritmust futtatok a rendezésre vagy legkisebb k kiválasztásra, hanem blokkonként 256 soros algoritmust párhuzamosan.

Az aktuális sorokhoz tartozó k értékeket regiszterekben tárolom. Azért esett erre a választás, mert a teljes regiszter fájl mérete Fermi előtt 16384*32 bit=64KB, ami jóval nagyobb, mint a megosztott memória 16 KB-ja. Mivel minden távolsághoz a címkét is tárolni kell így (16384 bájt)/(256 szál)/(2)/(4 bájt)=8, tehát maximum k=8-ra lehetne lefuttatni ezt a megosztott memória használatával.

Ezzel szemben a regiszteres módszerem k=21-re is működik regiszter spilling nélkül. A CUDA implementáció elkészítéséhez Java nyelven írtam egy kódgenerátort, ami létrehozza a megfelelő számú változót, és jelentős számú if-else feltétel, illetve for ciklus árán a beszúrásos rendezéshez hasonló módon rendezi a regiszterekben tárolt értékeket. A módszer sebességét jelentősen visszafogja a nagymértékű divergencia, tehát a sok if feltétel, illetve annak elágazásai.

De ennél sokkal nagyobb mértékben gyorsít az a tényező, hogy a szálaknak nem kell a globális memóriához fordulniuk jelentős késleltetéssel, ha az adott szálhoz tartozó ideiglenes k legkisebb érték tömb elejére egy új értéket kell beszúrniuk, ehhez ugyebár a tömb összes elemét egyel „hátrébb” kell mozgatni.

Ez a néhány órajel alatt írható és olvasható regiszterekben sokkal gyorsabban elvégezhető, mint a több száz alatt olvasható globális memóriában. A Fermi architektúrában kétszer annyi regiszter van, de az Nvidai fordító fejlesztő csapatával való levelezésemből megtudtam, sajnos van egy hardveres korlát, hogy egy szál hány regisztert használhat, alapvetően afelett regiszter spilling lép fel, ennek tesztelését még jelenleg is végzem. Így hogy ne k=21 legyen a felső korlát implementáltam egy beszúrásos rendezést, ami hasonló „*soros algoritmusokból sokat párhuzamosan*” elven működik.

5.4 A MEGFELELŐ CÍMKE KIVÁLASZTÁSA, ÁLTALÁNOS MŰKÖDÉS

A k-NN utolsó lépésénél csupán az az újítás, hogy ez is a GPU-n történik, habár elenyésző idő ezt végrehajtani, de a GPU-val ezen az időn is tudunk spórolni. Ezzel együtt felhívnám a figyelmet arra, hogy a teljes algoritmust a GPU-n számoltam ki, tehát a bemenetet az indításkor rámásoltam a kártyára és onnan már csak a végső eredményt másoltam vissza.

Az általános működéssel kapcsolatos fontos tényező, hogy ha lehetőség van rá, tehát a távolságmátrix számítás kimenete belefér egy textúrába, akkor azt kihasználva a CUDA 4.0 új funkcióját lineáris globális memóriából azonnal beolvasom textúrába, így csökkentve a legkisebb k és címke keresés futási idejét.

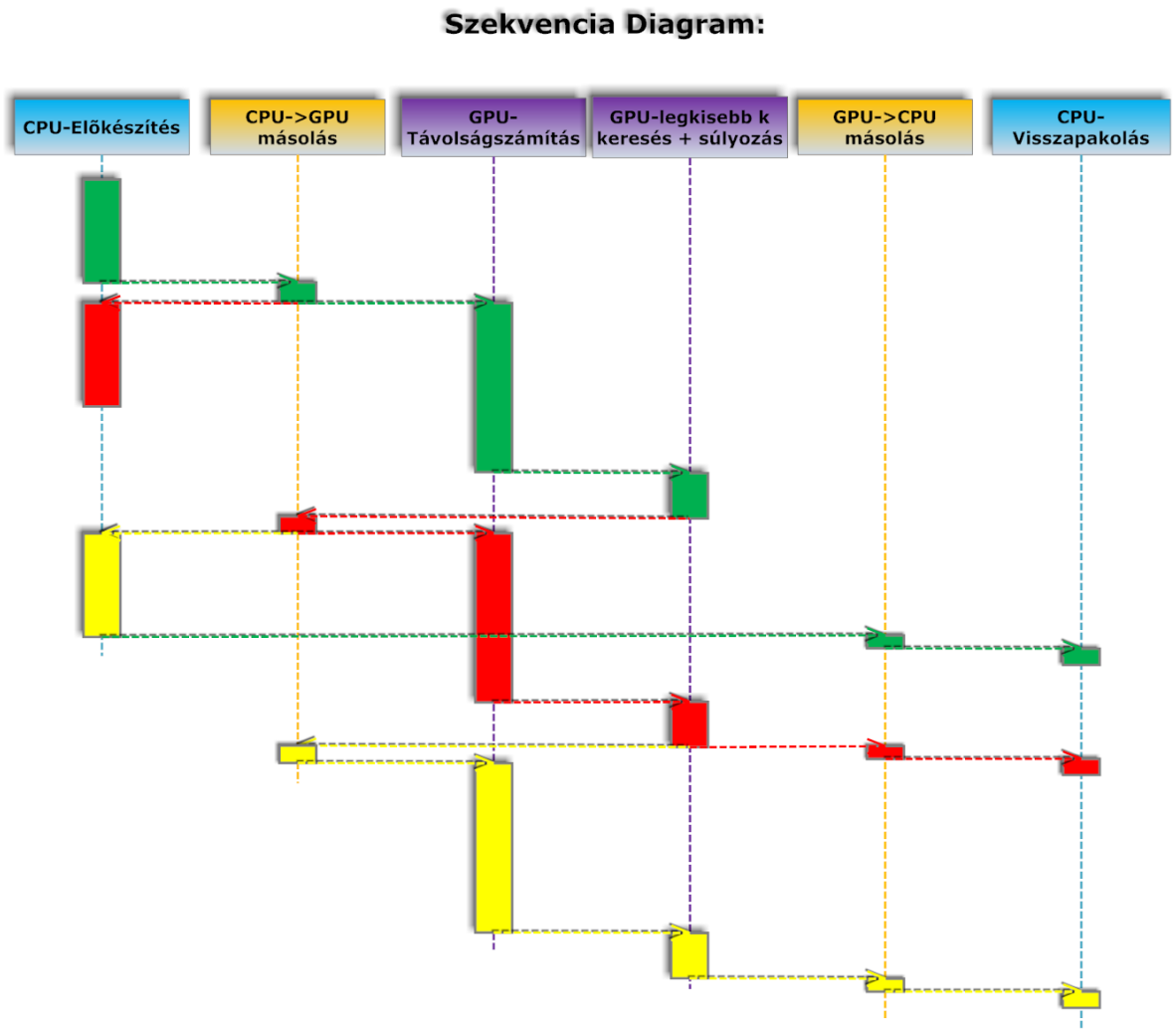
5.5 A PROBLÉMA FELDARABOLÁSA, ASZINKRON MŰKÖDÉS:

A korábbi megoldások egyik tanulsága, hogy a problémát fel kell bontani részproblémákra különben a GPU memóriája szűk keresztmetszetnek fog bizonyulni. A legnagyobb limitációja a programomnak az, hogy a tanító halmaznak el kell férnie a GPU memóriájában, hiszen ezen megkötés nélkül a tanító halmazt is folyamatosan mozgatni kellene a GPU és a CPU között, ami meglehetősen sok memóriát igényelne CPU oldalon is. Az algoritmus tulajdonsága, hogy ha a probléma egy részletét ki szeretnénk számítani, tehát például csak a teszt halmaz felére akarjuk először lefuttatni akkor szükségünk van a teljes tanító halmazra, hiszen ha nincs meg a teljes távolságmátrixunk, akkor a legkisebb k keresést se tudjuk végrehajtani.

De ezen relatív súlyos limitáción túl a program nagyon jól skálázódik, tehát ha a tanító halmaz mellett már nincs sok hely, akkor akár párszor 10 sor finomságra, vagy még az alá is fel tudja bontani a teszt halmazt.

Az eddigi implementációk komoly hiányossága, hogy nem használták a CPU-t miközben futottak, ami habár a GPU sokkal gyorsabb, de egy hatalmas veszteség, hiszen a párhuzamosság mértékét rontották ezzel. Ennek tudtában úgy terveztem meg a bővítményemet, hogy minél nagyobb mértékben képes legyen a CPU és a GPU számításokat átfedni, sajnos ebben a JCUDA is nehézséget jelentett, mivel kereséseim alapján előttem még senki nem próbálkozott JCUDA-ban aszinkron műveletek végrehajtásával. C-ben ezek teljesen egyértelműek, elindításuk után az irányítás azonnal visszakerül a CPU-hoz, Java-ban ez sajnos nem volt ilyen egyszerű, végül egy „cuStreamQuery" nevű a stream státuszát lekérdező függvény mellékhatásaként sikerült elérnem, hogy a feldolgozás aszinkron módon induljon.

Az aszinkron működés az alábbi ***32. ábrán*** figyelhető meg. Elsőként a lilával jelölt GPU-n futó kódrészletre hívnám fel a figyelmet, jól megfigyelhető a magas kihasználtság.



32. ÁBRA A PROGRAM FUTÁSÁNAK SZEKVENCIA DIAGRAMJA

Az ábrán azonos színűek az egy részproblémához tartozó műveletek, látható, hogy a GPU feldolgozással párhuzamosan történik CPU-n a következő részprobléma előkészítése, tehát kiolvasása a RapidMiner-ből, illetve az azt megelőző részprobléma eredményének lemásolása a GPU-ról, majd a részeredmény visszarakása a programba. Ennek köszönhetően a RapidMiner

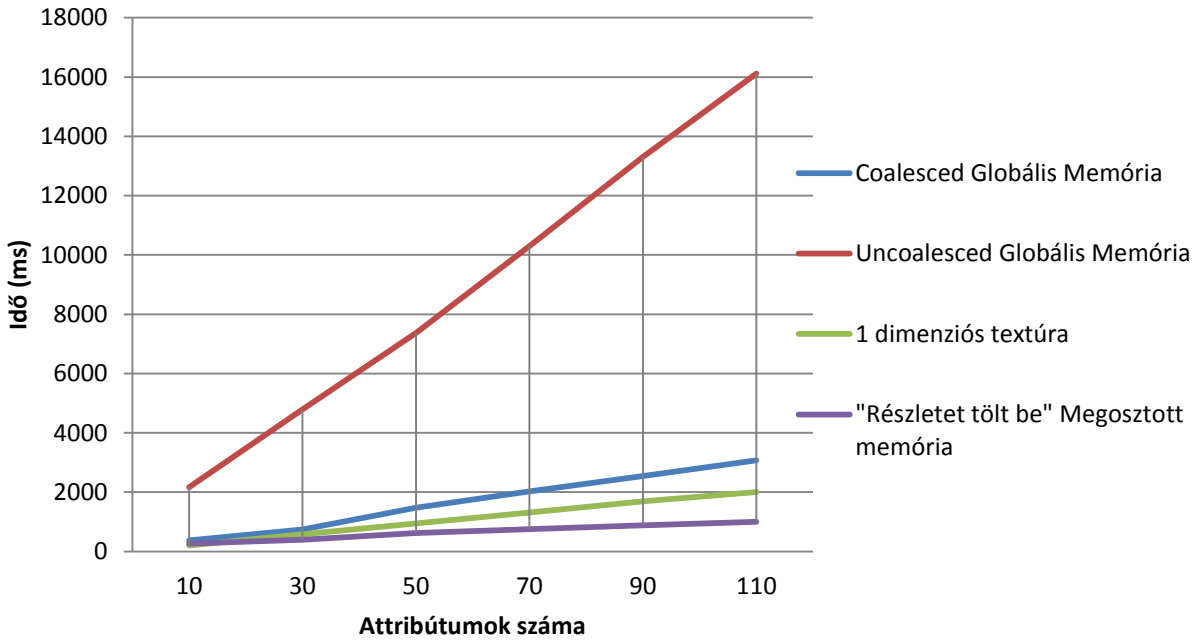
nem érzékeli, hogy a kód a GPU-n fut, a kompatibilitás magas a többi CPU-n futó operátorral, hiszen a bővítmény teljesen szabványos kimenetet ad. A következő fejezetben bemutatom mennyire gyorsan teszi ezt.

6. MÉRÉSEK

6.1 TÁVOLSÁGMÁTRIX SZÁMÍTÁS KÍSÉRLETI SZÁMÍTÁSOK

6.1.1 FERMI ELŐTTI ESZKÖZÖN

Az első mérésemmel néhány korábban említett implementációs technikával felhívom a figyelmet az uncoalesced globális memória hátrányára. Az ***1. diagramon*** 10000x10000es mátrixot számoltam *euklideszi távolsággal*, változó attribútum számmal.



1. DIAGRAM IMPLEMENTÁCIÓS TECHNIKÁK ÖSSZEHASONLÍTÁSA, EUKLIDESZI TÁVOLSÁG

Érdekes jelenség látható az ***1. táblázatban***, a tesztek során használt 1.2-es compute capability-s Geforce GT 335m kártyám maximum 24,1 GB/s-os memória sávszélességét jelentősen kihasználó globális memória technika sebességben milyen mértékben marad alul a „részletet tölt be” megosztott memóriás módszerhez képest.

A globális memóriás módszer közvetlenül ***coalesced*** módon olvassa a memóriát, és onnan végzi el a távolságszámítást, majd az eredményt visszaírja, a megosztott memóriás módszer a korábban ismertetett módon működik.

A lassulás oka a korábban említett kernel limitációkban keresendő, a globális módszert a sávszélesség limitálja a megosztott memóriásat pedig a maximális utasításszám. A méréseket Visual Profiler 4.0-val készítettem.

1. TÁBLÁZAT KERNELEK ÁLTAL ELÉRT SÁVSZÉLESSÉGEK

Attribútumok száma:	10	30	50	70	90	110
„Globális memória” sávszélesség (GB/s):	17,67	20,07	20,68	21,07	21,47	21,52
„Megosztott memória” sávszélesség (GB/s):	2,05	3,92	4,10	4,73	5,18	5,49
Sávszélesség különbség:	8,60	5,12	5,05	4,45	4,14	3,92
Gyorsulás a Globális Memóriához képest:	1,40	1,86	2,37	2,69	2,90	3,06

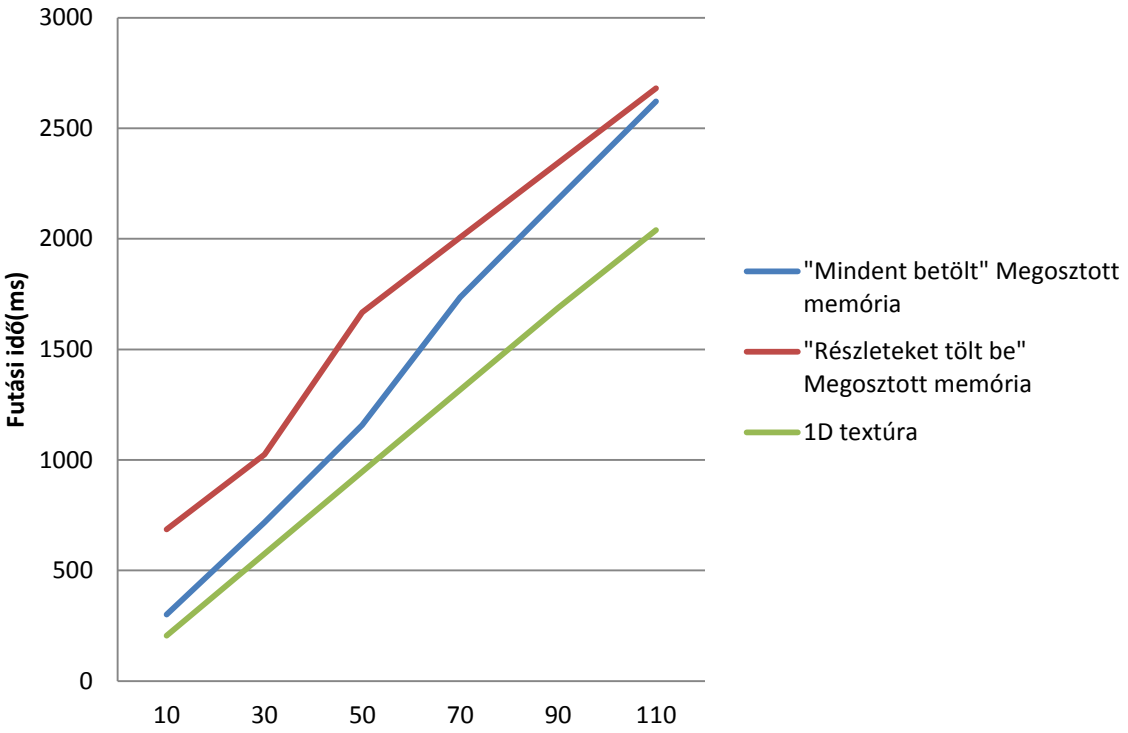
Ahogy említettem a megosztott memóriás módszert az utasítások száma limitálja, így két cache-elési stratégiát is implementáltam. A ***2. táblázatban***, a korábban már részletesen leírt elméletemet támasztom alá, tehát, hogy a „részletet tölt be” stratégia jelentősen több műveletet hajt végre, aminek köszönhetően a kernelek által elért globális memória sávszélességi is arányosan változik.

2. TÁBLÁZAT "MINDENT BETÖLT" ÉS "RÉSZLETET TÖLT BE" MÓDSZER ÖSSZEHASONLÍTÁSA

Attribútumok száma:	10	30	50	70	90	110
"Mindent betölt" sávszélessége (GB/s):	1,82	2,16	2,20	2,05	2,09	2,12
"Mindent betölt" utasításszáma:	8,39E+07	1,96E+08	3,10E+08	4,20E+08	5,31E+08	6,37E+08
"Részletet tölt be" sávszélessége (GB/s):	0,80	1,51	1,53	1,77	1,94	2,05
"Részletet tölt be" utasításszáma:	1,89E+08	2,82E+08	4,59E+08	5,51E+08	6,44E+08	7,35E+08
Sávszélességek aránya:	2,27	1,43	1,44	1,16	1,07	1,03
Végrehajtott utasítások aránya:	2,26	1,44	1,48	1,31	1,21	1,16

Fermi előtt, a megosztott memória mérete 16384 bájt. Egyszerre 16 teszt és 16 tanító elem összes float típusú attribútumát kell betöltenünk oda, így a maximális attribútum szám 128, de ezen érték mellett csak egy blokk futhat multiprocesszoronként, tehát az occupancy értéke nagyon alacsony.

A ***2. diagramon*** két módszer teljesítménykülönbsége figyelhető meg. A „mindent betölt” módszer az attribútum szám növekedésével, tehát az occupancy csökkenésével arányosan veszít az előnyéből. Emellett látható, hogy a textúra cache-ből történő olvasás *kevert euklideszi távolság* esetén teljesítménynövekedést jelent ezen architektúrán. Ezen diagramnál hívom fel a figyelmet arra, hogy összehasonlítva ezen diagramot az első diagrammal, ami az *euklideszi távolságot* mutatta észrevehető, hogy a két kernel az implementációik hasonlósága ellenére nagyon eltérő tulajdonságokkal rendelkeznek, látható ez például a textúra által okozott teljesítménynövekedésből.



2. DIAGRAM KEVERT EUKLIDESZI TÁVOLSÁG FERMI ELŐTT

Ez a jelenség jobban megfigyelhető a ***3. táblázatban***, ahogy az occupancy szintje csökken a gyorsulás is csökken. Érdekes jelenség, hogy még 256 szál és minimális occupancy esetén se lesz lassabb ez a módszer.

3. TÁBLÁZAT "MINDENT BETÖLT" MÓDSZER TULAJDONSÁGAI

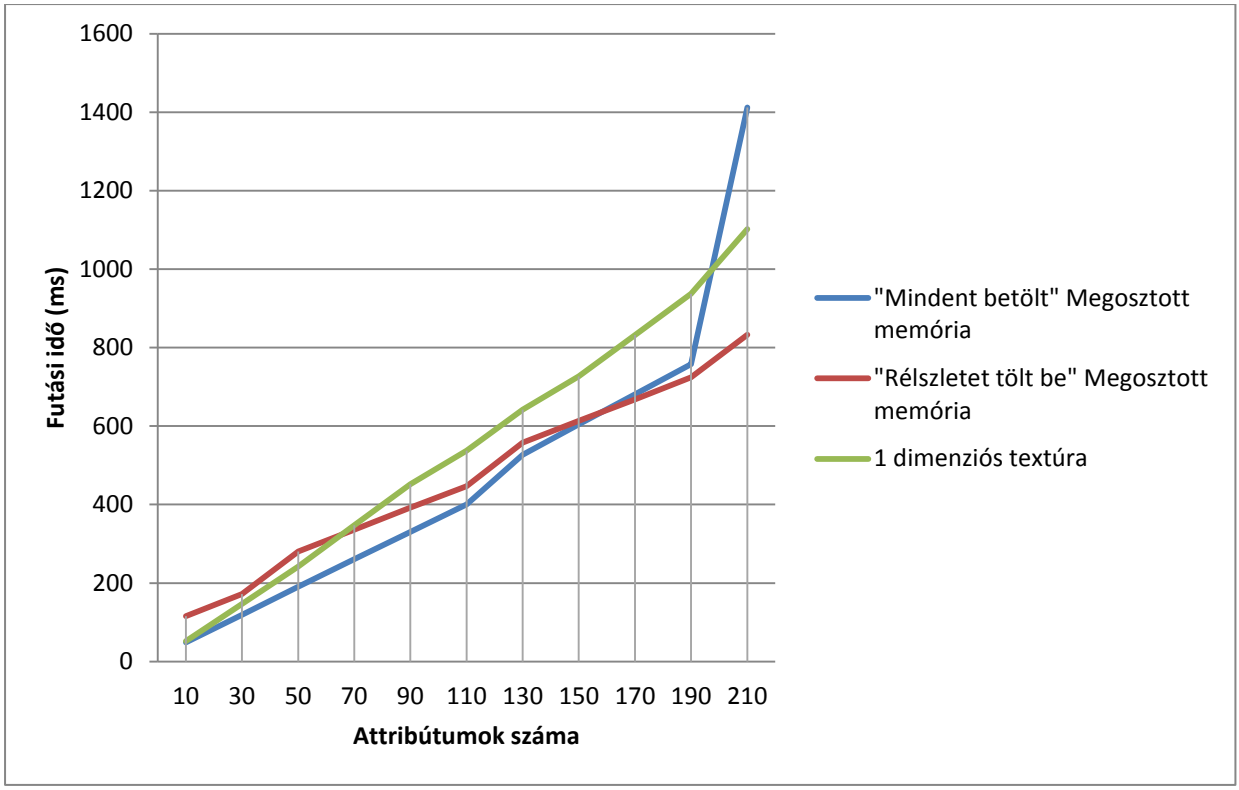
Attribútumok száma	**10**	**30**	**50**	**70**	**90**	**110**
Occupancy értéke:	1	1	0,5	0,25	0,25	0,25
Gyorsulás (%):	127,9070	42,6184	44,1659	15,6196	7,4771	2,2892

6.1.2 FERMI ARCHITEKTÚRÁN

Az előző pontban említett jelenség érdekesebb egy Geforce 480 GTX típusú, Fermi architektúrájú GPU-n. Látható a ***3. diagramon***, hogy az attribútum szám növekedésének köszönhetően csökken az occupancy és így veszít a teljesítményből a „mindent betölt” kernel.

Sőt mivel itt már 48 KB megosztott memória áll rendelkezésünkre tehát több attribútum fér el benne. 190 attribútum felett az is megfigyelhető, hogy az occupancy olyan alacsony szintre esik, hogy nem lesz a multiprocesszoron elegendő warp a globális memória késleltetések elfedésére, így a teljesítmény drasztikusan csökken.

Felhívom a figyelmet arra is, hogy mint látható, itt a textúra a korábbival ellentétben nem tud teljesítménynövekedést elérni.



3. DIAGRAM KEVERT EUKLIDESZI TÁVOLSÁG FERMIN

Az alábbi ***4. táblázatban*** jól látható az occupancy hatása a „mindent betölt” és a „részletet tölt be” stratégia közötti relatív teljesítményre, látható, hogy 110 és 130 attribútum között esik az occupancy olyan szintre, ahol a módszer elveszti a sebességelőnyét. Az „occupancy calculator” nevű eszköz segítségével meghatározva a pontos érték 128 attribútum, ami felett az occupancy 0.5-ről 0.33-ra csökken. Később ezt az értéket használom a két kernel közötti választásnál.

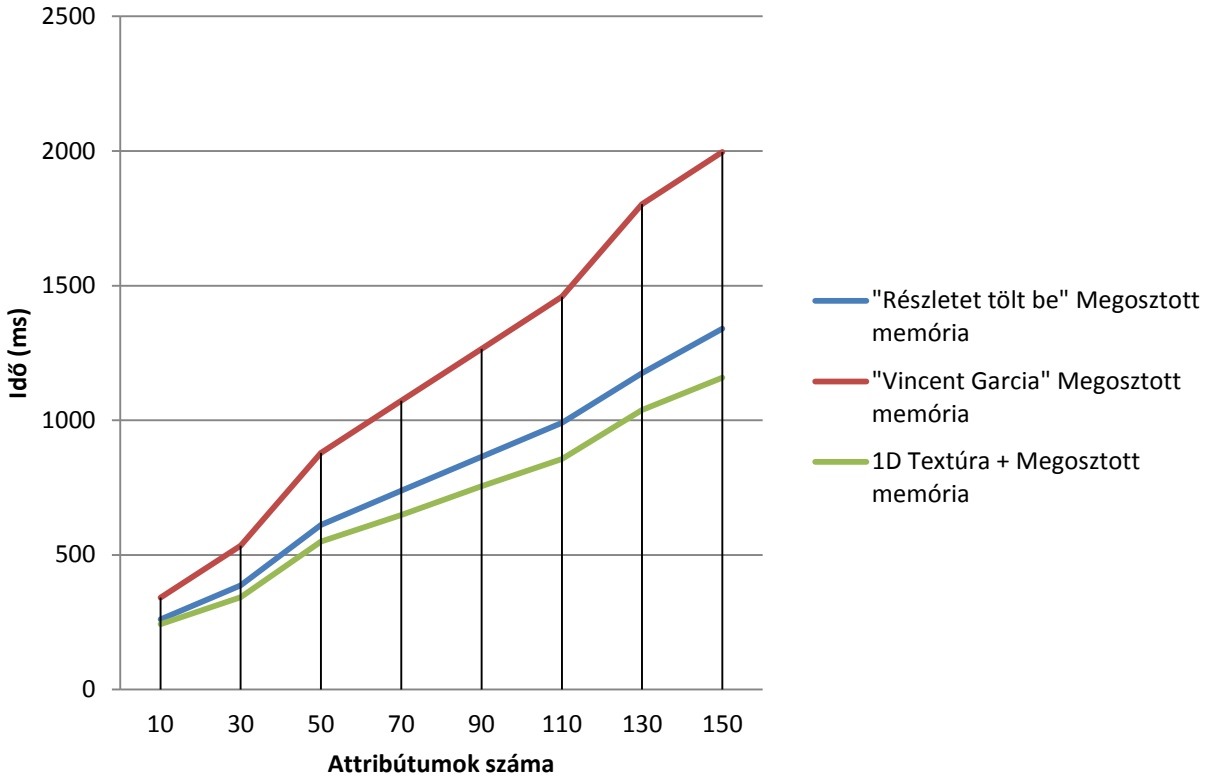
4. TÁBLÁZAT OCCUPANCY VÁLTOZÁS FERMI ARCHITEKTÚRÁN

Attribútumok száma:	**10**	**50**	**70**	**90**	**110**	**130**	**150**	**170**	**190**	**210**
Occupancy értéke:	1	1	0,83	0,67	0,5	0,33	0,33	0,33	0,33	0,17
Gyorsuáls(%):	136,73	46,60	28,74	18,79	11,75	5,88	1,49	-1,91	-4,49	-41,01

6.2 ÖSSZEHASONLÍTÁS A KORÁBBI MEGOLDÁSSAL

A 3 cikk írói közül egyedül Vincent Garcia[*1*] osztotta meg a kódját, így vele hasonlítom össze a korábban bemutatott módszereket. A ***4. diagramon*** látható, hogy a „részletet tölt be” megoldásom már önmagában jelentős teljesítménykülönbséget produkál a cikkben szereplő megoldáshoz képest, pedig alapvetően ugyanazon az elven alapulnak. A „mindent betölt” módszer csak *kevert euklideszi távolságnál* jelent előnyt, itt pedig a cikkben is szereplő egyszerű *euklideszi távolság* van, ezért itt nem alkalmazom.

Meglepő, de ha nem a globális memóriából, hanem textúrából olvasok a megosztott memóriába, hiába volt az teljesen coalesced, teljesítménynövekedést érek el. Ennek implementálásakor a kisebb korlátozásokkal rendelkező egy dimenziós textúrát és a „részletet tölt be" megosztott memóriás stratégiát ötvöztem, a textúra és megosztott memória egyidejű használata alatt később is ezt értem.



4. DIAGRAM KORÁBBI MEGOLDÁS VIZSGÁLATA FERMI ELŐTTI ARCHITEKTÚRÁN

Mélyebb vizsgálatból látható, hogy amíg az én implementációm bank-konfliktus mentes, ez a cikk írójának munkájáról nem mondható el, ez jelentős sávszélesség csökkenést is okoz.

5. TÁBLÁZAT KORÁBBI MEGOLDÁS HIÁNYOSSÁGÁNAK OKA

Attribútumok száma	10	30	50	70	90	110
Bank-konfliktus(ezer):	56 694	56 112	55 187	55 728	56 888	57 351
Sávszélesség (Vincent Garcia):	1,46	2,80	2,84	3,27	3,56	3,77
Sávszélesség (Saját megosztott):	2,11	4,00	4,17	4,81	5,26	5,57
Gyorsulás Textúrával (%):	40,33	55,69	60,11	65,43	67,55	70,44

Az ***5. táblázatból*** látható, hogy Fermi előtti architektúrán akár 70%-al is gyorsabb lehet a megoldásom, ugyanez az összehasonlítás Fermin is látható, a ***6. táblázatban***, ugyebár ott mivel a bankok száma kétszeresére nőtt már nem jelentenek olyan jelentős problémát a konfliktusok, de ezzel is akár 40% feletti gyorsulás is elérhető.

6. TÁBLÁZAT KORÁBBI MEGOLDÁS TELJESÍTMÉNYTESZTJE FERMI ARCHITEKTÚRÁN

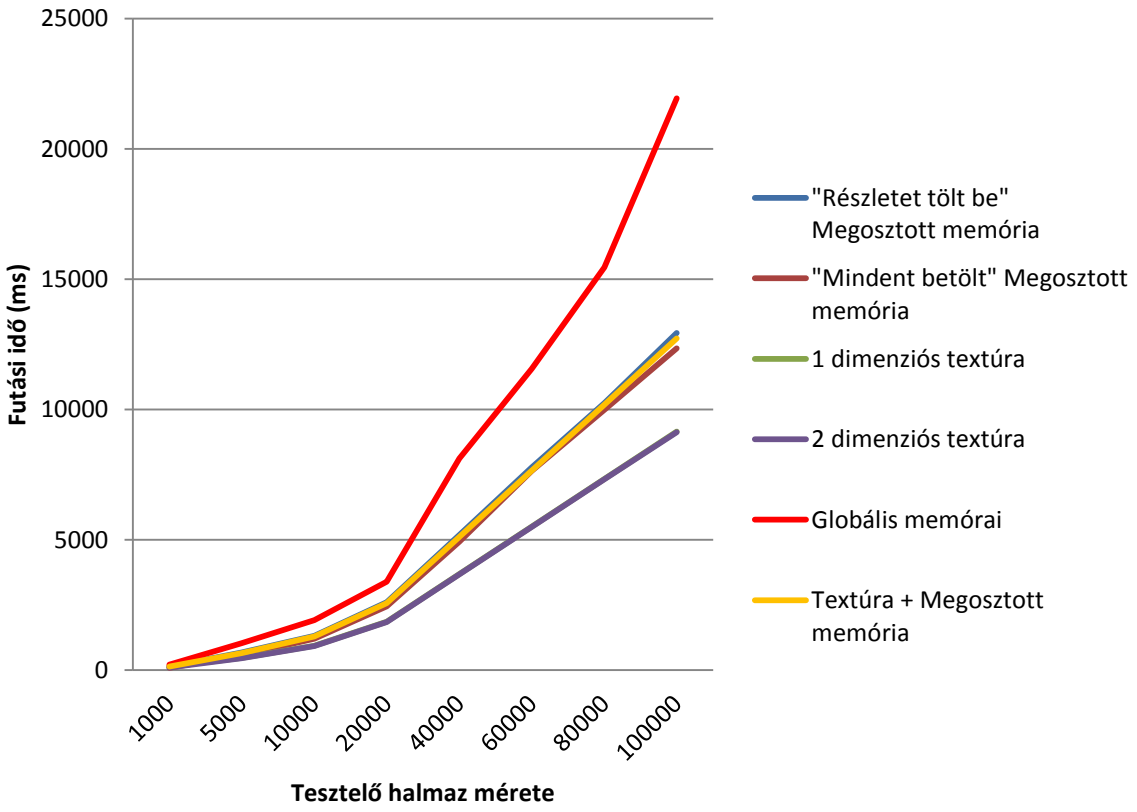
Attribútumok száma	10	30	50	70	90	110	130	150	170
"Vincent Garcia"	33	47	85	105	126	146	185	205	226
"Részletet tölt be"	32,1	46	73	88	101	116	146	160	177
Gyorsulás (%):	*2,80*	*2,17*	*16,44*	*19,32*	*24,75*	*25,86*	*26,71*	*28,13*	*27,68*
1D Textúra + Megosztott memória	31	44	69	82	95	107	133	145	158
Gyorsulás (%):	*6,45*	*6,82*	*23,19*	*28,05*	*32,63*	*36,45*	*39,10*	*41,38*	*43,04*

A korábbi teszteket 10000x10000 méretű távolságmátrixszal végeztem, mivel „Vincent Garcia” megoldását maximum ennyivel sikerült futtatnom.

6.3 VÉGLEGES IMPLEMENTÁCIÓK TESZTJE

Korábban csupán az implementációkat mértem külön, ezen pontban a bővítményben szereplő végleges módszereket fogom összehasonlítani.

Elsőként a *kevert euklideszi távolságot* mutatom be, miként reagál a két architektúrán.

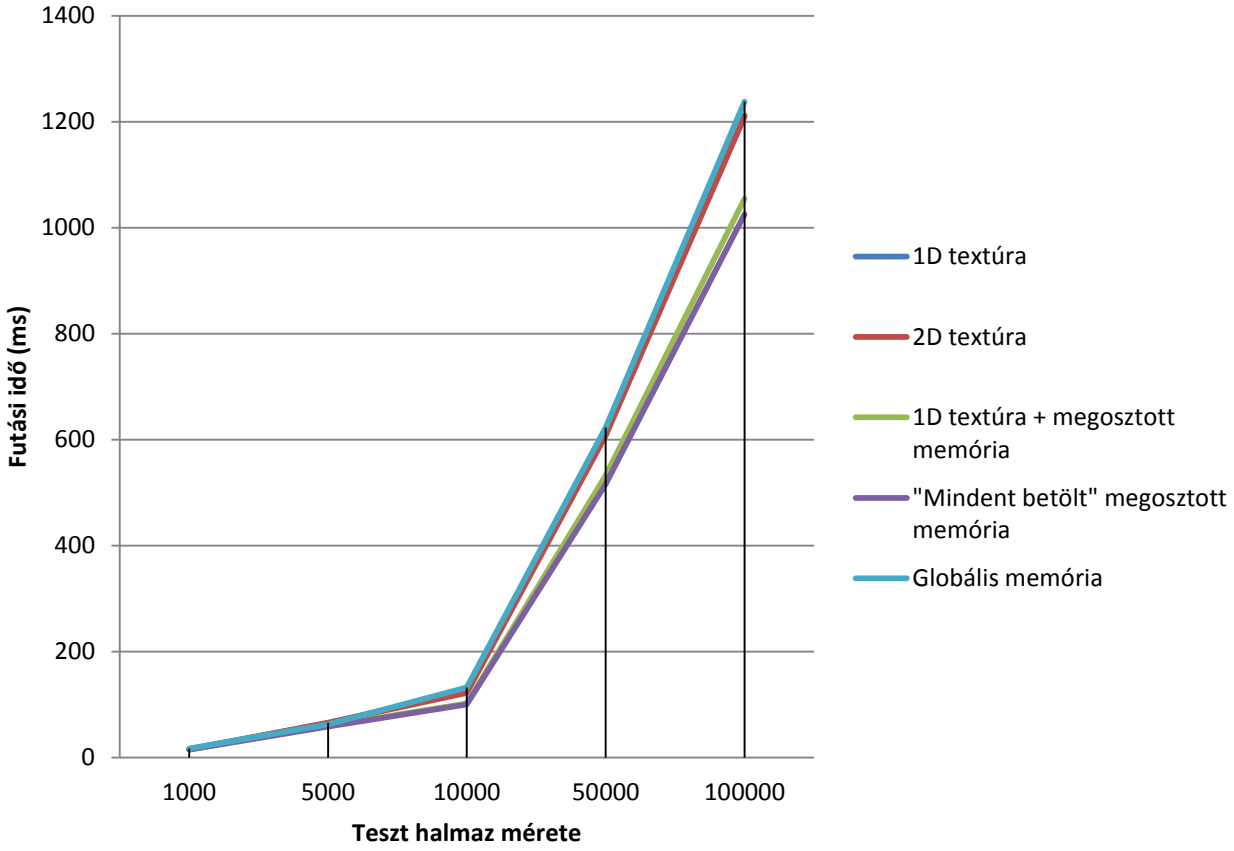


5. DIAGRAM KEVERT EUKLIDESZI TÁVOLSÁG FERMI ELŐTTI ARCHITEKTÚRÁN

Az ***5. diagramon*** megfigyelhető, hogy Fermi előtt a 2 dimenziós textúra nyújtja a legjobb teljesítményt, tehát ebből következik, hogy ez lesz a futtatás alapbeállítása, azonban ennek

mérete limitált, maximum (64x1024)x(32x1024), efelett már más technológiára kell váltani, különben a kernel nem fog lefutni.

Fermin ugyanezt lefuttatva a ***6. diagramon*** látható eredményt kapjuk. Mivel a globális memória is cache-elve van így a textúra már nem minden esetben tud teljesítménynövekedést okozni, ehelyett a különböző megosztott memóriás módszerek veszik át a vezető szerepet.

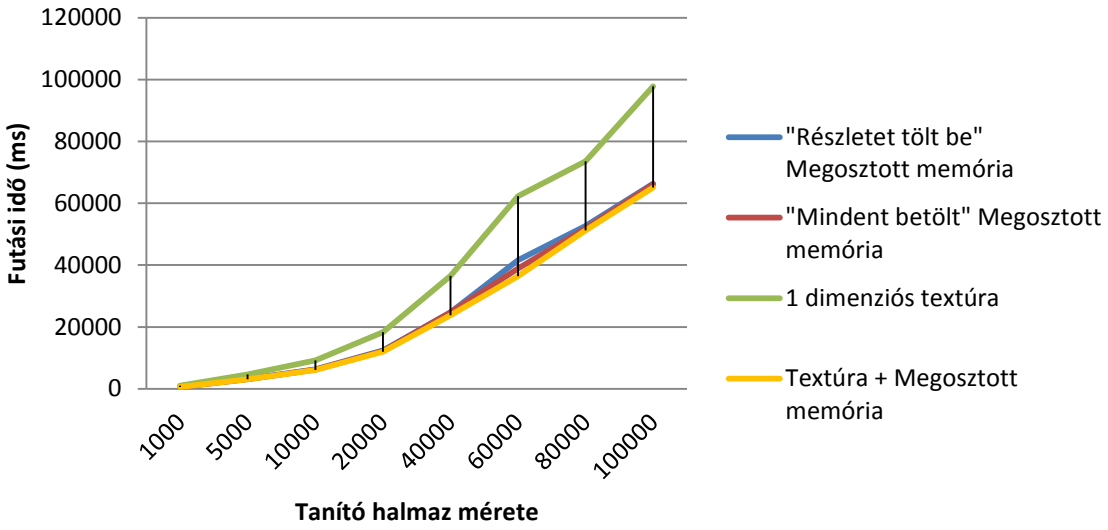


6. DIAGRAM KEVERT EUKLIDESZI TÁVOLSÁG FERMI ARCHITEKTÚRÁN

Az ennél kevésbé számításigényes *euklideszi távolságnál*, a Fermi előtti architektúráknál is más a helyzet, itt a megosztott memóriás módszerek teljesítményben túlszárnyalhatják a textúrát, ez megfigyelhető a ***7. diagramon***.

Valószínűleg ennek a kisebb számításigény az oka, tehát a maximálisan elvégezhető utasítások száma nem jelent olyan komoly limitációt, mint *kevert euklideszi* esetben, így a kernel több műveletet tud végrehajtani, több memóriaolvasást tud kezdeményezni, így jobban kihasználhatja a memória busz sávszélességét, ennek következtében összességében jobb teljesítményt tud elérni, mint az alapvetően memória sávszélesség által limitált kernel, amit a textúra cache-el gyorsítottunk.

A Fermin történő euklideszi távolság számítás implementációs karakterisztikája érdekes módon szinte teljesen megegyezik a ***7. diagramon*** láthatóval, csupán a sebesség nőtt jelentősen, így ennek közlésétől eltekintek.

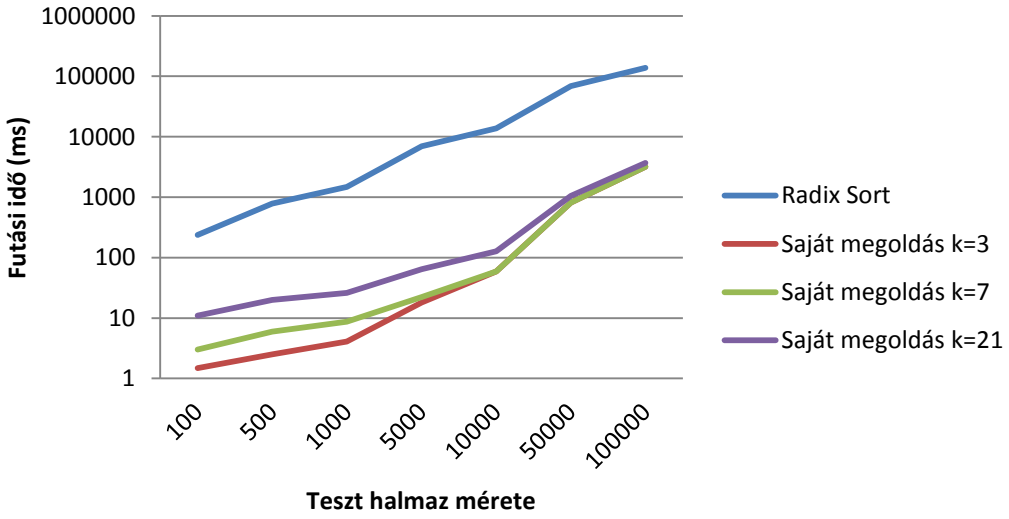


7. DIAGRAM EUKLIDESZI TÁVOLSÁG FERMI ELŐTTI ARCHITEKTÚRÁN

6.4 LEGKISEBB K ELEM KIVÁLASZTÁSÁRA ADOTT MÓDSZER MÉRÉSE

A [*2*] cikk ezt a feladatot úgy oldja meg, hogy a távolságmátrix minden sorára lefuttat egy párhuzamos rendező algoritmust, a Radix Sort egy korábbi párhuzamosított változatát. Majd egyszerűen minden sor elejéről az első k elemet használja csak fel. Én ehelyett a jelenleg elérhető leggyorsabb CUDA rendező implementációval hasonlítottam össze a megoldásomat, ami szintén Radix rendezés elvén működik. Ezen pont méréseit csak Fermin végeztem el.

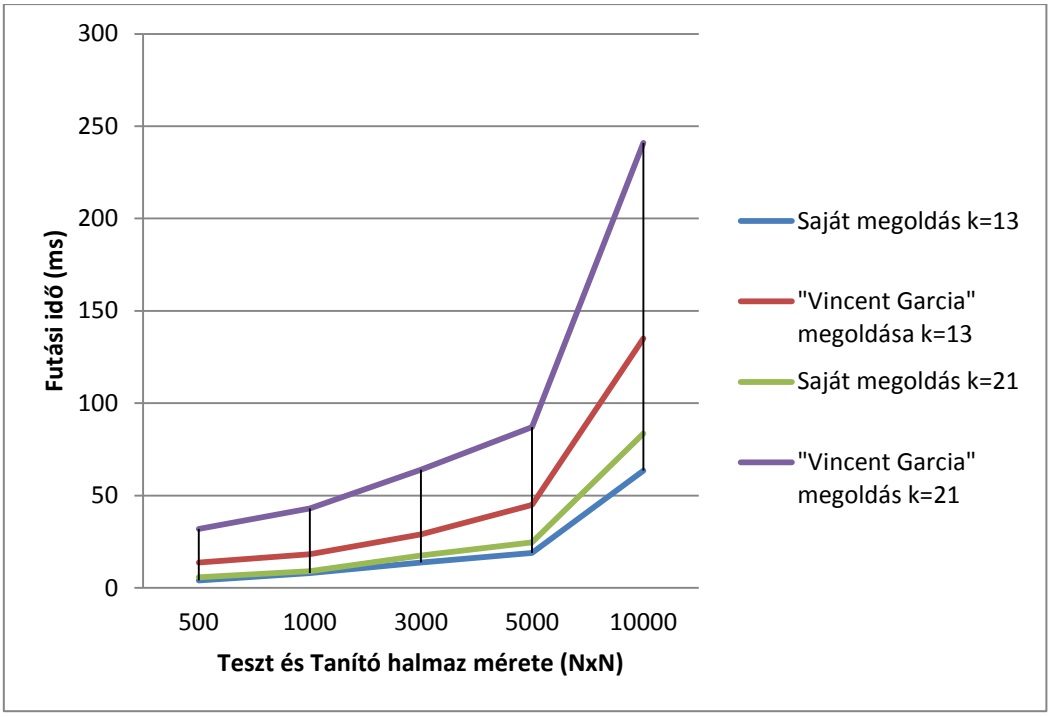
Habár a Radix rendezés egy sort, meglehetősen gyorsan rendez, de minden teszt elemhez külön sor tartozik, ezeknek a mérete a ***8. diagramon*** látható mérésnél 90000 elem, tehát ezt a rendezést annyiszor kell végrehajtani, mint ahány eleme van a teszt halmaznak, ez önmagában algoritmikailag is pazarlás hiszen, nem kell a teljes halmazt rendezni csupán a legkisebb k elemre vagyunk kíváncsiak.



8. DIAGRAM RADIX RENDEZÉS ÉS A SAJÁT REGISZTEREKEN ALAPULÓ IMPLEMENTÁCIÓM ÖSSZEHASONLÍTÁSA

Felhívom az olvasó figyelmét, hogy a ***8. diagram*** futási idő skálája kivételesen logaritmikus léptékű, így látható, hogy a gyorsulás láthatóan két nagyságrendű.

Vincent Garcia cikkének implementációjához az enyém nagyon hasonló elven működik, minden sorra egy soros legkisebb k elem kiválasztót futtat, csak a nagy különbség abban rejlik, hogy ő ezeket a k értékeket illetve a hozzájuk tartozó címkéket a globális memóriában tárolja, én pedig regiszterekben helyezem el őket. A kernelem teljesítményét két hatás befolyásolja egyik az, hogy a kernel divergenciája nagy, sokszor kell a warp-okat újrafuttatni más if-else feltétel ág mentén, de emellett regiszterekben végzi el azt, amit a másik implementáció a globális memóriában, ennek következtében, mint a ***9. diagramon*** látható jelentősen gyorsabb, mint a korábbi cikk megoldása. Megfigyelhető az is, hogy a globális memória késleltetése miatt a nagyobb k értékekre egyre nagyobb a különbség a két implementáció között.



9. DIAGRAM LEGKISEBB K KERESŐ MÓDSZEREK ÖSSZEHASONLÍTÁSA

6.5 KIEGÉSZÍTŐ FUNKCIÓKKAL KAPCSOLATOS MÉRÉSEK

A bővítmény robusztusságának és minőségének javítása érdekében néhány kiegészítő funkciót is implementálnom kellett, az ezekről készült rövid méréseket ebben a pontban mutatom be. Ezen pont méréseit sajnos már nem volt időm Fermi architektúrán is elvégezni, de a bemutatott tulajdonságok nagy valószínűséggel ott is érvényesek.

6.5.1 A HIÁNYZÓ ÉRTÉKEK KEZELÉSE

A követelmények között szerepelt ezen értékek kezelésének lehetővé tétele, mivel mint látszani fog ez bizonyos esetekben meglehetősen erőforrás-igényes lehet, ezért ezt opcionálisan kikapcsolhatóvá tettem. Ennek hatékony megoldásához egy külön kernelt kellett létrehoznom, hiszen ha ezt elágazásként implementálom, akkor a teljesítményre az még drasztikusabb hatással lehetett volna. A ***7.*** és a ***8. táblázat*** eredményeit összehasonlítva látható, hogy főként az euklideszi távolság teljesítményét fogta vissza ez a művelet. Euklideszi esetben a megosztott

memóriát és textúrát kombináló, kevert esetben pedig az egy dimenziós textúrán alapuló módszert alkalmaztam, tehát az adott tartományban a lehető leggyorsabbat.

7. TÁBLÁZAT NAN ELLENŐRZÉS EUKLIDESZI TÁVOLSÁG ESETÉN

Távolságmátrix mérete (NxN):	5000	10000	50000	100000
NaN ellenőrzés nélkül:	282	770	16689	66904
NaN ellenőrzéssel:	311	910	24690	98756
Többletidő (%):	10,284	18,182	47,942	47,609

8. TÁBLÁZAT NAN ELLENŐRZÉS KEVERT EUKLIDESZI TÁVOLSÁG ESETÉN

Távolságmátrix mérete (NxN):	5000	10000	50000	100000
NaN ellenőrzés nélkül:	245	961	26540	91468
NaN ellenőrzéssel:	305	998	28482	101305
Többletidő (%):	24,490	3,850	7,317	10,755

6.5.2 ASZINKRON MŰKÖDÉS ELŐNYEI

Az 5.5 pontban bemutattam az aszinkron működés szekvencia diagramját, ennek a sebességre gyakorolt hatása a ***9***. illetve a ***10. táblázatban*** látható.

9. TÁBLÁZAT ASZINKRON MŰKÖDÉS IDŐNYERESÉGE EUKLIDESZI TÁVOLSÁG ESETÉN

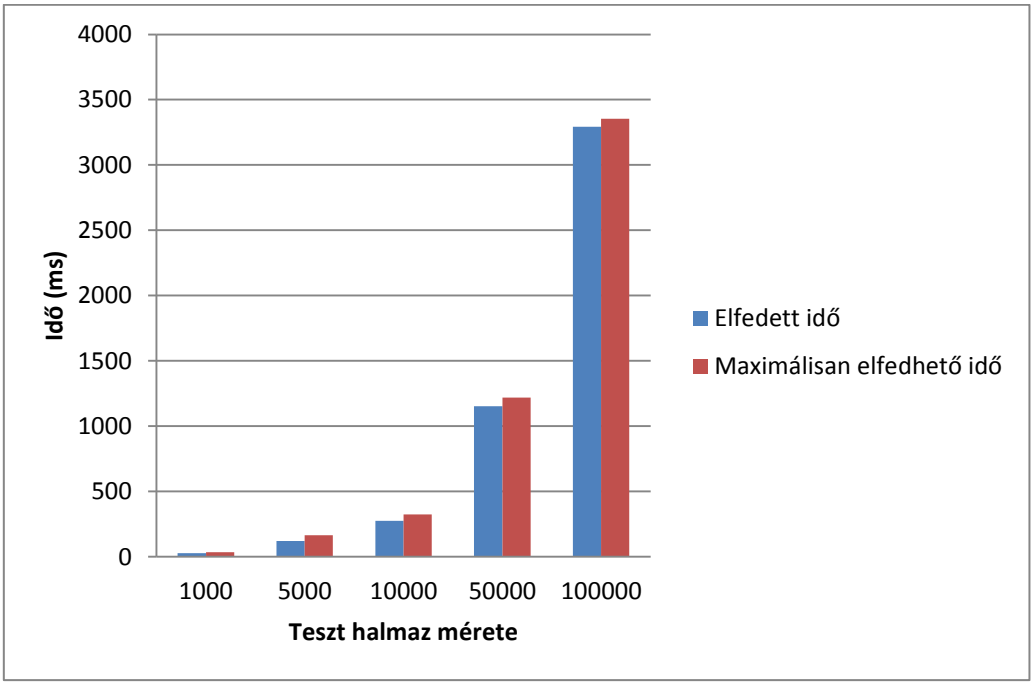
Távolságmátrix mérete (NxN):	5000	10000	50000	100000
Aszinkron működés:	235	726	16018	62900
Szinkron működés:	282	770	16689	66904
Időnyereség (%):	20,00	6,06	4,19	6,37

Megfigyelhető a ***10. táblázatban***, hogy a *kevert euklideszi távolság* esetén, aminek nagyobb a számításigénye jobban át lehet lapolni a CPU előkészítési, feldolgozási illetve a GPU számítási idejét.

10. TÁBLÁZAT ASZINKRON MŰKÖDÉS IDŐNYERESÉGE KEVERT EUKLIDESZI TÁVOLSÁG ESETÉN

Távolságmátrix mérete (NxN):	5000	10000	50000	100000
Aszinkron működés:	232	908	20112	76632
Szinkron működés:	245	998	26864	91468
Időnyereség (%):	5,60	9,91	33,57	19,36

Azon kérdésre, hogy ezen értékek mennyire hatékonyak, tehát a CPU feldolgozásának mekkora részét tudjuk ténylegesen elfedni, illetve átlapolni a GPU-val a ***10. diagram*** ad választ. A maximálisan elfedhető időbe, sajnos nem tartozik bele a GPU-ra történő másolás, mert ennek aszinkron módon JCUDA-ban történő hatékony megvalósíthatósága, mint korábban említettem kérdéses. Tehát a maximálisan elfedhető idő, az adatok előkészítése a GPU-ra való másolás előtt, a GPU által kiszámolt eredmények visszamásolása a CPU-ra és azok visszarakás a RapidMiner-be. Az elfedett idő alatt, egy kernel szinkron és aszinkron futtatása közötti időkülönbséget kell érteni, mint látható, ilyen szemléletben ez a pár százalék nyereség is hatékonynak nevezhető.



10. DIAGRAM KEVERT EUKLIDESZI TÁVOLSÁG ASSZINKRON MŰKÖDÉSÉNEK HATÉKONYSÁGA, A TANÍTÓ HALMAZ MÉRETE 10000 ELEM

6.5.3 KEVERT EUKLIDESZI TÁVOLSÁG FUTÁSÁNAK BIZTOSÍTÁSA NAGY ATTRIBÚTUM SZÁM ESETÉN

Már írtam arról, hogy a *kevert euklideszi távolság* esetén, azt hogy az attribútumok nominális vagy numerikus típusúak a konstans memóriában tárolom, mert ott a leghatékonyabb, de ennek mérete sajnos korlátos. Így maximum 16384 attribútumra

működhetne a kernel, ami bizonyos esetekben kevés lehet, ezért ha ennél több attribútumra van szükség, akkor a típus értékeket is a textúra memóriából olvasom ki, ennek a teljesítményre gyakorolt negatív hatása a ***11. táblázatban*** látható.

11. TÁBLÁZAT AZ ATTRIBÚTUM TÍPUSÁNKA KIOLVASÁSA KÉT KÜLÖNBÖZŐ MEMÓRIATÍPUSBÓL

Távolságmátrix mérete (NxN):	**5000**	**10000**	**50000**	**100000**
Típus a textúra memóriából:	358	1390	34239	137151
Típus a konstans memóriából:	244	931	22850	91472
Időtöbblet (%):	46,72	49,30	49,84	49,94

6.6 ÖSSZEHASONLÍTÁS A CPU-S MEGFELELŐVEL

A bővítményemet két számítógép RapidMiner-ébe telepítettem, hogy bemutassam a relatív sebesség növekedést, ezeket a ***12.*** illetve a ***13. táblázatban*** lehet látni. Mindkét esetben kevert euklideszi távolságot futattam az adott tartományban legoptimálisabb technológiával. A ***12. táblázatban*** mivel 50 attribútumról van szó, így a „mindent betölt" megosztott memóriás módszer, a ***13. táblázat*** esetén pedig az egy dimenziós textúra eredménye szerepel. Habár ***13. táblázatban*** szereplő gyorsulások alacsonyabbak a Fermin tapasztaltaknál, ez nem az optimalizáció hibája, ez főként annak köszönhető, hogy a Fermis konfiguráció GPU/CPU teljesítmény aránya jóval kedvezőbb volt.

12. TÁBLÁZAT FERMI ARCHITEKTÚRÁJÚ GPU ÖSSZEHASONLÍTÁSA 4 MAGOS CPU-VAL

Távolságmátrix mérete	**1000x1000**	**5000x5000**	**10000x10000**	**50000x50000**	**100000x100000**
Geforce 480 GTX GPU idő (s):	0,306	0,822	1,673	8,1	20,65
Intel Quad Core q8400 CPU idő (s):	1	11	62	1334	3533
Gyorsulás:	3,268	13,382	37,059	164,691	171,090

13. TÁBLÁZAT FERMI ELŐTTI ARCHITEKTÚRÁJÚ LAPTOP GPU ÉS 2 MAGOS LAPTOP CPU

Távolságmátrix mérete	**1000x1000**	**5000x5000**	**10000x10000**	**50000x50000**	**100000x100000**
Geforce GT 335m GPU	0,073	0,188	0,908	27,702	81,138
Intel i5 450m CPU idő (s):	0	1	54	1452	4198
Gyorsulás:	~1	5,319	59,471	52,415	51,739

Feltűnő lehet, hogy egy bizonyos méretnél nem futattam nagyobb teszteket, ennek két fő korlátja volt, egyik az idő, a másik a CPU memória, ami mindkét konfigurációban 4 GB volt, ami mai körülmények között átlagosnak nevezhető. Sajnos ennek csak töredékét lehet a Java virtuális géphez rendelni, ez pedig ellehetetlenítette, hogy ténylegesen nagy méterű fájlokkal

teszteljek, ennek ellenére a futtattam néhány tesztet nagyméretű fájlokra, ezek eredménye a ***14. táblázatban*** *látható*. Ekkor azonban a tanító halmaz méretét le kellett vennem kicsire, hogy elférjek a CPU memória Java virtuális gépnek átadott területébe. A futási időkből is látszik, hogy a memória volt a korlát.

14. TÁBLÁZAT FUTTATÁS NAGY FÁJLOKRA

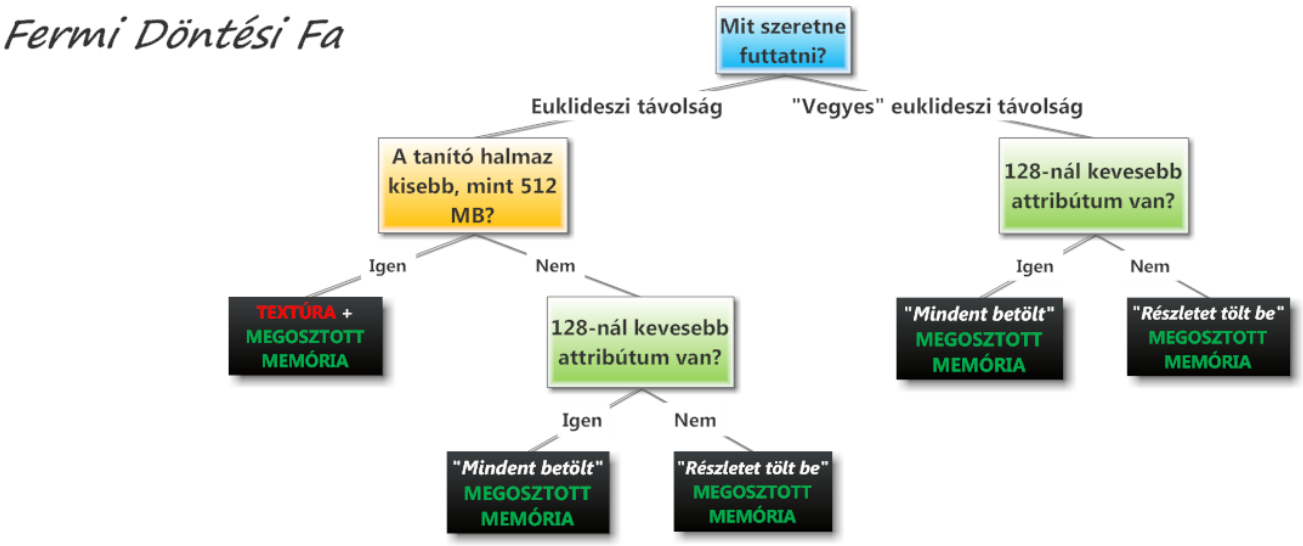
Tesztelő halmaz mérete:	500000	1000000	3000000
Futási idő (s):	51,6	97,4	439,1
Fájl mérete (MB):	192	385	1151

6.7 A KÉSZ PLUGIN TECHNIKÁK KÖZÖTT VÁLASZTÓ DÖNTÉSI FÁI:

Ezen pont az egyik legfontosabb részét mutatja be a bővítménynek, azt, hogy a korábban bemutatott implementációk közül melyiket milyen esetekben alkalmazza, ezek természetesen a mérési eredményeken alapulnak.

A ***33. ábrán*** figyelhető meg a Fermi döntési fája, mint látható erre az architektúrára alapvetően három implementációs módszert alkalmazó kerneleket kellett írnom. *Euklideszi távolság* esetén azt kell leellenőrizni, hogy a tanító halmaz belefér-e az egy dimenziós textúra maximális méretébe, ez 2^{27} elemet jelent, ami lebegőpontos értékek esetén maximum 512 MB. Ez nem tűnhet komoly korlátnak azonban a speciálisan GPGPU-ra épített Nvidia Tesla kártyáknak akár több mint 12 GB globális memóriája is lehet, ezen belül könnyedén előfordulhat ennél nagyobb halmaz. A tanító halmaz mérete nem jelent limitációt hiszen azt tetszőlegesen feldarabolhatjuk.

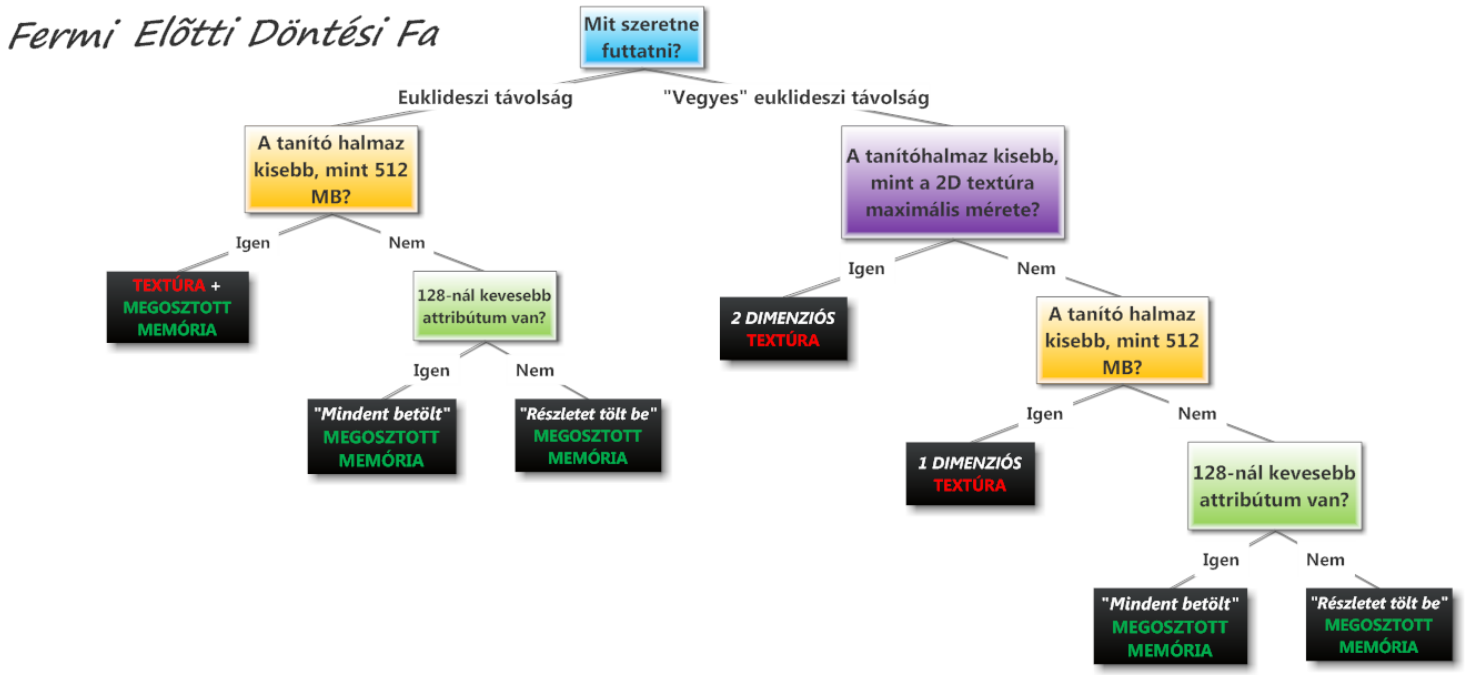
Ha belefér a textúrába, akkor textúrából olvas a megosztott memóriába, ha nem akkor egy olyan ág fut le, ami megegyezik a *kevert* vagy más fordítással *vegyes euklideszi távolságéval*. Ekkor az attribútum szám dönt, hiszen ez határozza meg, hogy melyik megosztott memóriás módszert érdemes használni. A 128 attribútum, ahogy korábban írtam egy vízválasztó, ami felett az occupancy 0,5-ről 0,33-ra esik.



33. ÁBRA FERMI IMPLEMENTÁCIÓ DÖNTÉSI FA

A ***34. ábrán*** látható, hogy a Fermi előtti architektúrákra való optimalizálás jóval összetettebb. Habár az *euklideszi távolság* ág ugyanúgy néz ki itt 128 attribútum felett nem az occupancy esése, hanem a megosztott memória maximális mérete jelent korlátot.

A *kevert euklideszi távolság* esetén pedig további két kernelt illesztettem a fába, hiszen látható volt, hogy mivel nincsenek a globális memória olvasások cache-elve, ezért teljesítménynövekedés érthető el textúrák használatával. Az is látható volt a teszteknél, hogy a két dimenziós textúra gyorsabb az egy dimenziósnál, így ott azt kell ellenőrizni, hogy a tanító halmaz belefér-e egy kétdimenziós textúrába, az egy dimenzióshoz hasonlóan a teszt halmazt pedig bele lehet darabolni.



34. ÁBRA FERMI ELŐTTI IMPLEMENTÁCIÓ VÁLASZTÓ DÖNTÉSI FA

Ezen két fa is jól reprezentálja, hogy a bővítmény sebességben és robusztusságban milyen mértékben túlmutat a korábbi megvalósításokon.

7. ÖSSZEFOGLALÁS, JÖVŐBELI TERVEK

Az elmúlt 8 hónapban megismerkedtem a CUDA nyelvvel, annak tényleges működésével és sikerült olyan kerneleket írnom, ami egy adott méretű bemenetre jól voltak optimalizálva. Ezeket gondos mérésekkel és tervezéssel, figyelembe véve a hardver adottságait, sikeresen összefűztem egy sokkal robusztusabb programmá, ami megfelelő hardver birtokában nem száll el, ha a felhasználó tízszer vagy százszor nagyobb bemeneten szeretne számolni. Majd megismerkedtem a JCUDA nyelvvel, amiben a CUDA legegyszerűbbnek tűnő funkcióiért is meg kellett küzdeni, tettem ezt azért, hogy napjaink egyik legnépszerűbb nyílt forráskódú adatbányászati szoftverébe be tudjam építeni a programom. Ezen keresztül sokkal több emberhez eljuthat és sokkal többen tudják használni.

Az elsődleges jövőbeli célom tehát a bővítmény letölthetővé tétele, ennek tervezett időpontja 2012 első negyedéve, addig azonban még további adatbányászati algoritmusokat is szeretnék megvalósítani.

Szeretnék köszönetet mondani az Irányítástechnika és Informatika Tanszék Számítógépes Grafika Csoportjának amiatt, hogy Fermi architektúrájú GPU-n tesztelhettem, enélkül munkám értéke valószínűleg jóval kisebb lenne.

IRODALOMJEGYZÉK

[1] Vincent Garcia, Eric Debreuve and Michel Barlaud: Fast k Nearest Neighbor Search using GPU, Universite de Nice-Sophia Antipolis, Sophia Antipolis, France, 2008

[2] Quansheng Kuang, and Lei Zhao: A Practical GPU Based KNN Algorithm, School of Computer Science and Technology, Soochow University China, 2009

[3] Shenshen Liang, Ying Liu, Cheng Wang and ˈLiheng Jian: A CUDA-based Parallel Implementation of K-Nearest Neighbor Algorithm, Chinese Academy of Sciences Beijing, China, 2009

[4] NVIDIA CUDA C Programming Guide Version 4.0 3/25/2011

[5] Yonghong Yan, Max Grossman and Vivek Sarkar: JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA, Department of Computer Science, Rice University, 2009

[6] Harvard University, CS 264 Massively Paralell Computing tárgy, diasor http://www.cs264.org/lectures/files/CS264_2011_03-CUDABasics_share.pdf - 2011.10.28.

[7] Harvard University, CS 264 Massively Paralell Computing tárgy, diasor http://www.cs264.org/lectures/files/CS264_2011_04-CUDAIntermediate_share_tmp.pdf - 2011.10.28.

[8] Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf - 2011.10.28.

[9] CUDA Shared Memory & Cache + Live Q&A With Dr Steve Rennich, NVIDIA - http://developer.download.nvidia.com/CUDA/training/sharedmemoryusage_july2011.mp4 - 2011.10.28.

[10] CUDA Warps And Occupancy Considerations+ Live With Dr Justin Luitjens, NVIDIA - http://developer.download.nvidia.com/CUDA/training/cuda_webinars_WarpsAndOccupancy.pdf - 2011.10.28.

[11]Vasily Volkov, UC Berkeley: Better Performance at Lower Occupancy - http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf - 2011.10.28.

[12] CUDA C BEST PRACTICES GUIDE v4.0 May 2011 - Design Guide

[13] Harvard University, CS 264 Massively Paralell Computing tárgy, diasor http://www.cs264.org/lectures/files/CS264_2011_05-CUDAAdvanced_share_tmp.pdf

[14] The CUDA Compiler Driver NVCC - Last modified on: <01-27-2011>

[15] Szénási Sándor: Óbudai Egyetem, GPGPU alapjai tárgy, - http://nik.uni-obuda.hu/app/APP03.pdf - 2011.10.28.

[16] http://www.kdnuggets.com/polls/2011/tools-analytics-data-mining.html - 2011.10.28.

[17] Approaching Vega: The final descent: How to extend RapidMiner 5.0

[18] CUDA Optimization: Identifying Performance Limiters By Dr Paulius Micikevicius - http://developer.download.nvidia.com/CUDA/training/cuda_webinars_dentifying_performance_limiters.pdf - 2011.10.28.

[19] http://data.hu/get/4360780/W41ZHH-KOVACS.pdf - 2011.10.28.