Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Hálózati Rendszerek és Szolgáltatások Tanszék

Szimplex algoritmus GPU-s megvalósítása IP hálózat forgalmi mátrixának becsléséhez

Berghammer Tamás

Konzulens:
Jakab Tivadar, HIT

2013

Tartalomjegyzék

1. Bevezetés

Jelenleg egy videokártya körülbelül 50-100-szor akkora számítási kapacitással rendelkezik, mint egy hasonló kategóriás CPU. A közeljövőben ennek az aránynak a videokártyák javára történő további eltolódása várható, mivel a hagyományos processzorok sebességének növekedése az utóbbi években gyakorlatilag megállt (a processzormagok számának növekedésétől eltekintve), a videokártyák esetén viszont a processzorok számának és ezzel együtt a számítási kapacitásnak további növekedése várható. A videokártyákban rejlő teljes számítási kapacitás kiaknázásához többek között arra van szükség, hogy egy probléma megoldása erősen párhuzamosítható legyen (sok ezer szálon) és a felhasznált adatok mennyiségéhez képest sok, de egyszerű műveletet kelljen végrehajtani.

A videokártyákban rejlő számítási kapacitások komoly lehetőségeket nyújtanak a műszaki problémák megoldása terén, mivel a GPU-k által nyújtott számítási kapacitás gyakorlatilag mindenki számára elérhető, aki viszonylag modern számítógépet használ. Jelenleg még aránylag kevés szoftver használja ki ezt a lehetőséget (a MATLAB például néhány mátrix művelethez kihasználja), ami azzal magyarázható, hogy csak az utóbbi években vált széleskörűen is elérhetővé és jól használhatóvá a videokártyák általános számításokra történő használatának lehetősége. A mérnöki gyakorlatban használt bonyolult szoftverek pedig csak viszonylag lassan változnak a nagy komplexitásból és a komoly megbízhatósági igényekből eredő magas fejlesztési költségek miatt.

Sok műszaki (és egyéb) probléma megfogalmazható lineáris programozási feladatként. Ezeknek a problémáknak a megoldása komoly számítási kapacitást igényel, ezért megfelelő algoritmus és implementáció választása esetén a GPU-k használatával a futásidő csökkenését lehet elérni. Ennek az előrelépésnek azért van fontos szerepe, mivel ugyanannyi idő alatt bonyolultabb modelleket használva is ki lehet számolni egy adott probléma megoldását, és ezzel az eredeti problémára egy pontosabb eredményt lehet adni.

A dolgozatom 2. fejezetében bemutatom az általános célú GPU programozás történetét és a benne rejlő lehetőségeket. Ezt követően a 3. fejezetben leírom egy IP hálózat forgalmi mátrixa meghatározásának problémáját, amit illusztrációként használok a szimplex algoritmus GPU alapú megvalósításának gyakorlati alkalmazására. A 4. fejezetben bemutatom a lineáris programozás GPU-val történő megvalósításának lehetőségeit, majd az 5. fejezetben felhasználom a lineáris programozás GPU-s implementálásával elért eredményeket a forgalmi mátrix meghatározásához. Végül a 6. fejezetben összefoglalom az elért eredményeket és felsorolom a további kutatási lehetőségeket.

A piacorientált kutatás fejlesztés támogatására a Nemzeti Fejlesztési Ügynökség pályázatán a Flexiton Információtechnológiai Kft. és a BME Hálózati szolgáltatások és Rendszerek Tanszéke által a Távközlő hálózatok középtávú tervezőrendszere (ARIADNE/MtP) kidolgozására közösen elnyert támogatás alapján (KMR_12-1-2012-0131) a GPU alapú megoldások hálózattervezési alkalmazásainak kidolgozása is folyik. Ennek a munkának egyik célja hatékony forgalombecslési módszerek kidolgozása egyszerű hálózati mérések eredményeire és azokat kiegészítő információkra alapozottan. A TDK dolgozatban bemutatott eredményeim ehhez az alkalmazásorientált kutatásfejlesztéshez kapcsolódnak.

2. Az általános célú GPU programozás

2.1. A GPGPU programozás és a CUDA története [12]

Az első videokártyák kifejezetten a grafikus megjelenítés felgyorsítását szolgálták és kizárólag az ehhez szükséges viszonylag bonyolult célfüggvényeket támogatták hardveres implementáció segítségével. Az 1990-es évektől viszont a videokártyák fejlődésével ezeket a függvényeket már nem kizárólag hardveresen valósították meg, hanem részben szoftveresen, aminek hatására a videokártyák programozhatóvá váltak. Az 1990-es évek végére már nemcsak a grafikusok és játékfejlesztők használták ki intenzíven a videokártyákban rejlő lehetőségeket, hanem a kutatók is felismerték, hogy a videokártyák nagyon hatékonyan használhatóak műveletek elvégzéséhez lebegőpontos számokon. Ekkor indult meg az általános célú GPU programozás, a GPGPU (General Purpose GPU) programozás.

Ebben az időben a GPGPU programozás meglehetősen bonyolult volt, mivel a fejlesztőknek a tudományos számításokat háromszögek és poliéderek segítségével kellett leírniuk ahhoz, hogy a GPU végre tudja őket hajtani. Ehhez szükség volt a grafikus API-k részletes ismeretére és grafikai programozásból származó tapasztalatokra is (például OpenGL programozás).

Az első komoly eredményt 2003-ban érte el egy Ian Buck által vezetett kutatócsoport, amikor a C nyelvhez kifejlesztettek egy fordítót, amelyik tartalmazott egy olyan kiegészítést, aminek a segítségével sokkal kényelmesebben lehetett GPU-ra általános célú programokat fejleszteni. A könnyebb használhatóságon kívül ezzel a fordítóval sokkal jobb futási sebességet lehetett elérni a fordítás közbeni optimalizációnak köszönhetően, mint a kézzel írt kódokkal.

Ezekre az eredményekre támaszkodva az NVIDIA Ian Buck vezetésével kifejlesztette a saját általános célú GPU programozási platformját, a CUDA-t (Compute Unified Device Architecture), amit 2006-ban mutatott be. Ez volt a világ első valóban általános célú videokártya programozási keretrendszere. Azóta már a 3.5-ös eszköz architektúránál és az 5.5-ös driver verziónál tartanak úgy, hogy minden egyes főverzió váltásnál új funkciók kerültek a keretrendszerbe.

A GPGPU és a CUDA jelenlegi erősségét mutatja, hogy a világ legnagyobb szuperszámítógépeiben a számítási kapacitás jelentős részét már a GPU-k nyújtják. Például a Titan (Oak Ridge, USA, jelenleg a világ második legerősebb szuperszámítógépe), teljes 27 PFLOPS-os elméleti teljesítményéből több mint 10 PFLOPS-ot már a GPU-k adnak. (PFLOPS: 10^{12} Floating Point Operation Per Second). A BME szuperszámítógépében pedig a teljes 6 TFLOPS-os teljesítményből összesen 2 TFLOPS-ot nyújtanak a videokártyák annak ellenére, hogy mindössze 4 videokártya van benne a 60 CPU mellett. A mérnöki gyakorlatban jelenleg sokkal kisebb a GPU-k jelentősége, ami elsősorban azzal magyarázható, hogy egy meglehetősen új (7 éves) technológiáról van szó, ezért még kevés szoftver támogatja. Mivel a CUDA képes videokártyák viszonylag olcsóak, ezért várható, hogy azokon a helyeken, ahol szükség van a komolyabb számítási kapacitásokra, hamarosan el fognak terjedni.

2.2. Jól párhuzamosítható problémák

A mai videokártyák számítási kapacitása sokszorosan meghaladja a hagyományos processzorok számítási kapacitását, ennek ellenére nem minden probléma oldható meg velük hatékonyabban. Ennek oka, hogy a legtöbb, hagyományos processzorokra kifejlesztett és optimalizált algoritmus nem tudja kihasználni a videokártyákban rejlő számítási kapacitást, mivel a legtöbb algoritmus ehhez nem (eléggé) párhuzamosítható.

2.2.1. A hatékony GPU-s implementálhatóság feltételei

Ha egy problémát a GPU képességeinek kihasználásával szeretnénk megoldani, akkor a legfontosabb elvárás az alkalmazott algoritmussal szemben, hogy egyszerre (nagyon) sok szálon lehessen végrehajtani, mivel a mai GPU-k teljesítményének kihasználásához legalább több tízezer, de sok esetben akár néhány millió szálra is szükség van. Ha ennek a feltételnek megfelel a felhasznált algoritmus, akkor sok esetben a videokártyán futó algoritmus végrehajtása gyorsabb lesz, mint ha ugyanazt az algoritmust CPU-n futtatnánk. A probléma ezzel a megközelítéssel az, hogy ha csak arra figyelünk, hogy elég sok szálon fusson az algoritmus, akkor annak ellenére, hogy a videokártya több százszor akkora számítási teljesítménnyel rendelkezik, mint a processzor, a futásidő csak kis mértékben lesz jobb.

Az elegendő szál létrehozása mellett a második legfontosabb dolog a nagy számítási sebesség eléréséhez a megfelelő memóriakezelés. A jelenleg használt CUDA kompatibilis videokártyákban többféle memória áll rendelkezésre. Ezek közül a 3 legfontosabb a globális memória, amelyik minden szál számára elérhető, a megosztott memória (shared memory), amelyiket az egy blokkon belül lévő szálak érik el, és a regiszterek, amelyek csak az adott szál számára hozzáférhetőek. Ezeken kívül még vannak speciális memóriák, mint például a konstans memória, de azoknak az alap szintű sebességoptimalizálás szempontjából kisebb a szerepük. A memóriák mérete a felsorolás sorrendjében csökken (általában 1-4 GB globális memória, blokkonként 32-64 KB megosztott memória, és blokkonként 32768-65536 darab regiszter), viszont a sebességük ezzel egyidőben nő. Ha egy algoritmus minden memóriaírásnál és -olvasásnál a globális memóriát használja, akkor a futási sebességet a memória hozzáférések nagyon erősen limitálják. Ezt a limitációt tovább növeli, ha a memória hozzáférések nem strukturáltan történnek.

A szálak száma és a memória hozzáférések optimalizálása mellett még két dolog befolyásolja jelentősen a teljes algoritmus futásidejét a bemenő adatok megkapásától a kimenő adatok kiírásáig. Az egyik az, hogy a GPU-n egy kernel [1] elindítása időbe telik. Az ehhez szükséges idő egészen rövid (általában kevesebb mint 1 ms), viszont ha sok, rövid kernelt futtatunk, akkor szignifikánssá válhat. A másik tényező, ami a teljes algoritmus futásidejét erősen befolyásolhatja, az a CPU memóriája és a GPU memóriája közötti adatmozgatások száma és a mozgatott adatok mennyisége. Ennek az az oka, hogy a videokártya egységnyi idő alatt sokkal több adatot tud feldolgozni, mint amennyit át lehet másolni a CPU memóriájából a GPU memóriájába abban az esetben, ha egy adattal, csak kevés műveletet kell végrehajtani. Az ilyen jellegű adatmozgatásokból származó időt sok esetben le lehet csökkenteni a teljes

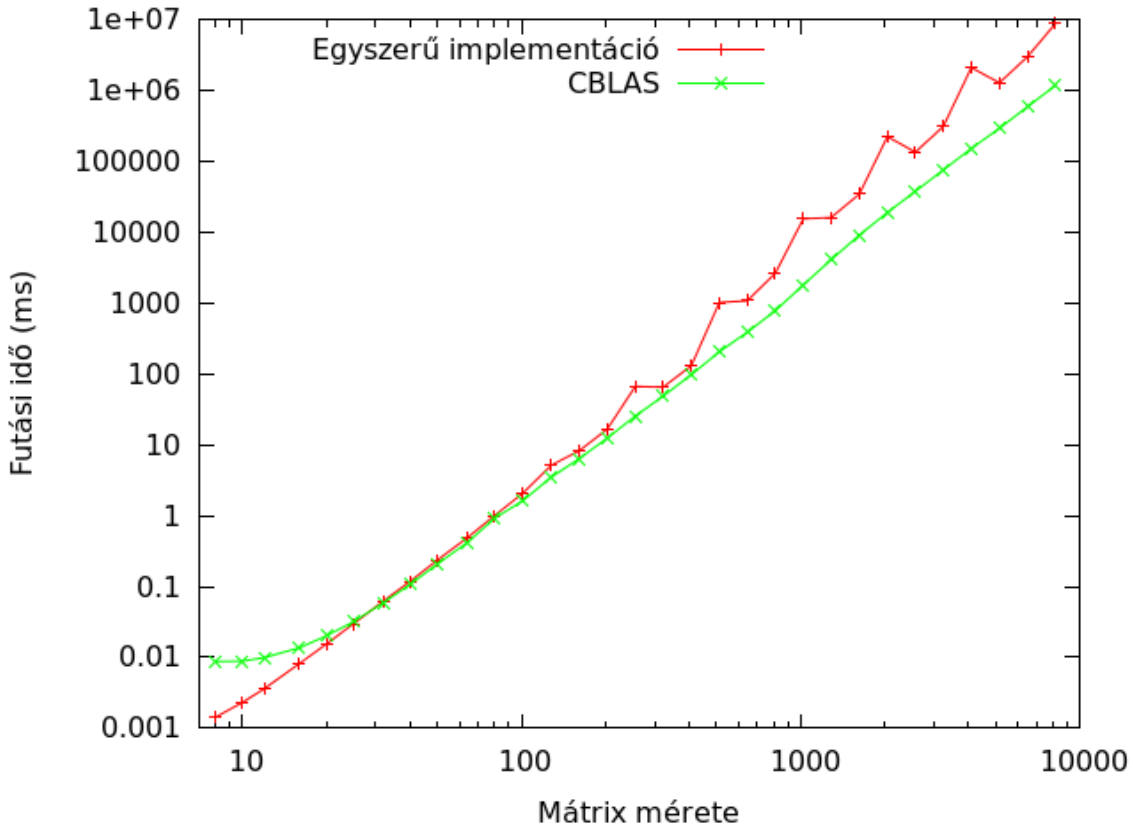
[1] kernel: GPU-n futtatott programrészlet

futásidőhöz képest abban az esetben, ha minden műveletet a videokártyán végzünk el, mivel akkor nem kell minden művelet előtt és után átmásolni az adatokat a CPU és a GPU memóriája között.

2.2.2. Mátrix szorzás

Az egyik legjobban párhuzamosítható és CUDA-ban hatékonyan implementálható probléma a mátrix szorzás. A továbbiakban megvizsgálok 5 különböző implementációt (2 CPU alapú és 3 GPU alapú) a futásidő szemszögéből, ezzel bemutatva, hogy mire képes egy GPU egy hasonló kategóriás CPU-hoz képest.

A két CPU-s implementáció közül az első a hagyományos $O(n^3)$-ös mátrix szorzó algoritmusnak a triviális implementációja C++-ban, a második pedig a GNU GSL-ben [4] szereplő CBLAS implementáció, amelyik egy sokkal bonyolultabb és hatékonyabb algoritmust valósít meg egy erősen optimalizált kóddal. (BLAS: Basic Linear Algebra Subprograms). A két algoritmus futásideje a 1. ábrán látható különböző méretű, négyzetes mátrixokra, ahol a mátrix minden eleme egy egyenletes eloszlású véletlen szám a [0;1) intervallumon.



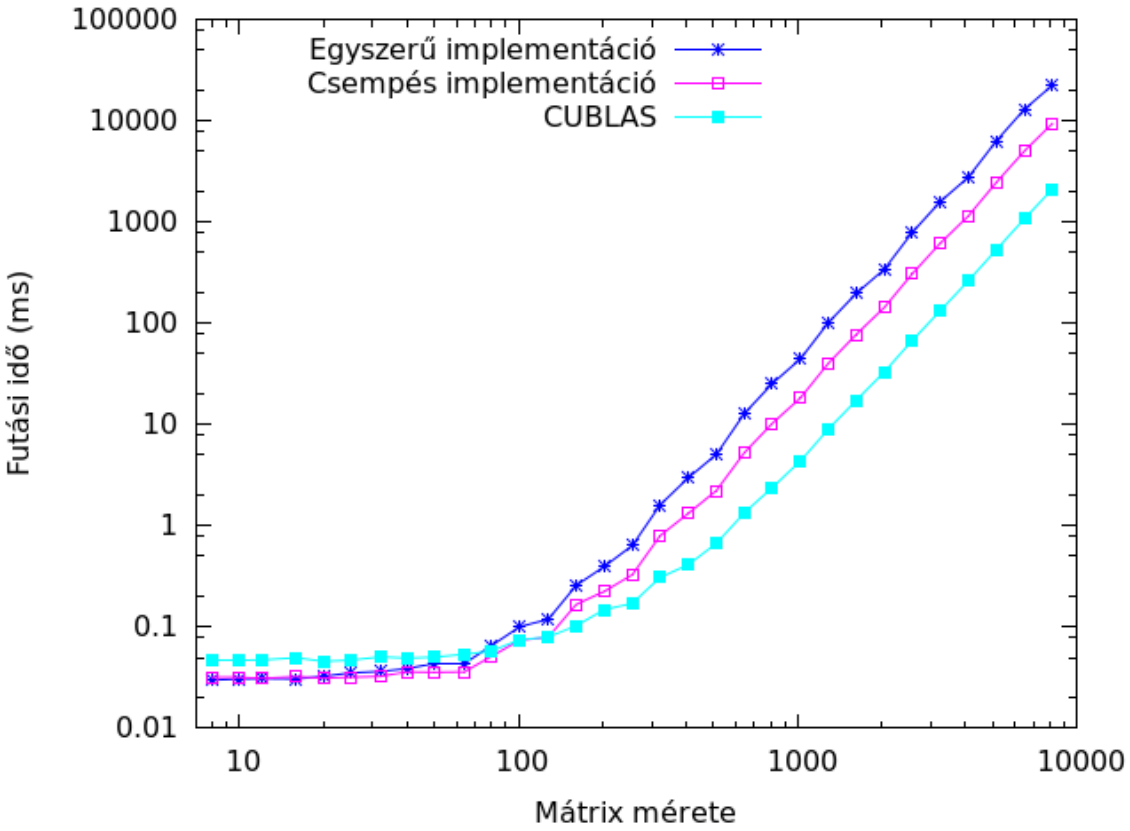
1. ábra. CPU alapú mátrix szorzó algoritmusok futásideje

A GPU-s implementációk közül az első a hagyományos $O(n^3)$-ös algoritmust valósítja meg a legegyszerűbb módon. Konkrétan úgy, hogy az eredmény mátrix minden egyes elemét egy külön szál számítja ki mindenféle memóriaolvasási optimalizálás nélkül.

A második GPU-s implementáció ugyanezt az algoritmust használja, viszont a globális memóriaolvasások számát drasztikusan lecsökkenti úgy, hogy a bemeneti mátrixokat $32 * 32$-es úgynevezett csempékre osztja, és minden egyes beolvasásánál egy teljes csempét beolvas a blokk szinten megosztott memóriába, és utána azt onnan olvassa ki az adott blokkban lévő összes szál. Ezzel a technikával a szükséges globális memóriaolvasások számát a 32-ed részére le lehet csökkenteni. Erre azért van szükség, mivel az első algoritmus futásidejét a memóriaolvasások limitálták.

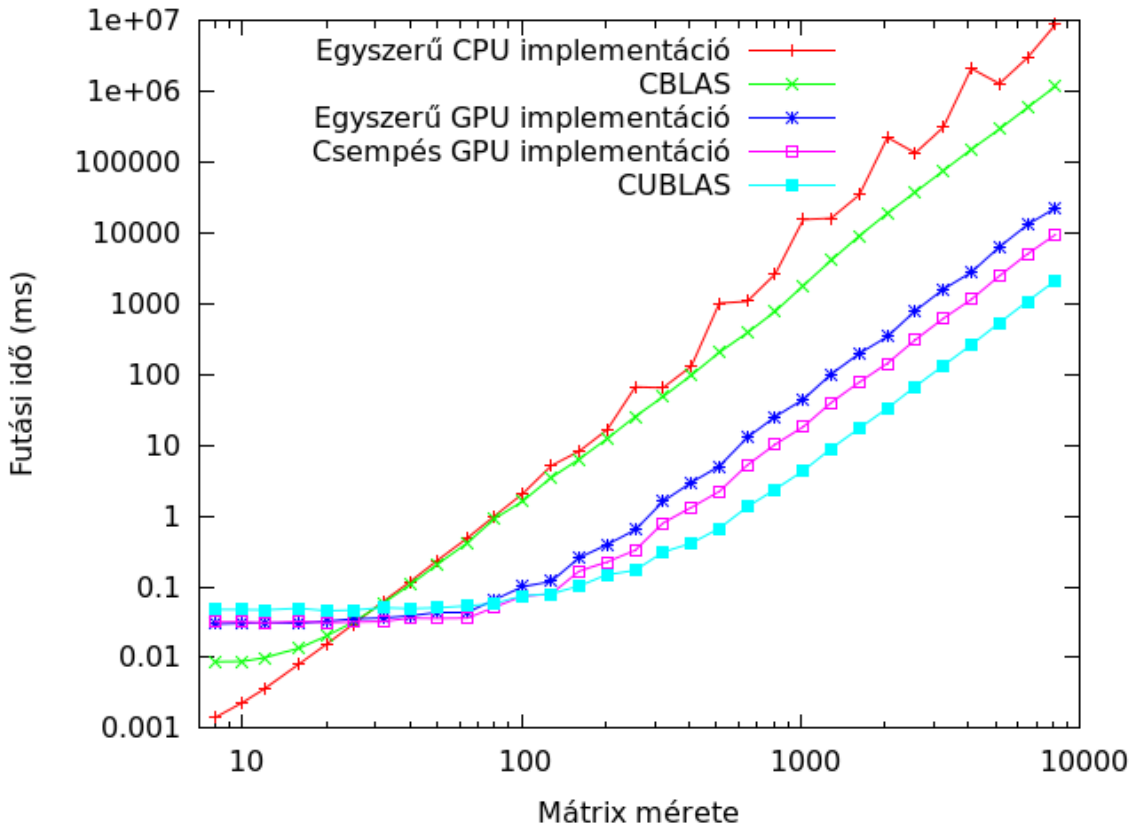
A harmadik GPU-s implementáció az Nvidia által írt CUDA SDK részét képező CUBLAS [11] osztálykönyvtárat használja a mátrix szorzás elvégzéséhez. Ez a CPU-s CBLAS-hoz hasonlóan egy hatékony (de bonyolult) algoritmust használ egy erősen optimalizált implementációval párosítva.

A három GPU-s algoritmus futásideje a 2. ábrán látható ugyanazokkal a bemeneti adatokkal, mint amit a CPU-s algoritmusok összehasonlításánál használtam. A grafikonról leolvasható, hogy a blokk szinten megosztott memória használata jelentős javulást eredményezett a futásidőben, illetve az is látszik, hogy a sokkal hatékonyabb algoritmust használó CUBLAS osztálykönyvtár sokkal gyorsabban működik, még a megosztott memóriát használó algoritmusnál is, viszont a különbség itt kisebb, mint a CPU esetén az alap implementáció és a GNU GSL-ben lévő CBLAS implementáció között volt. Ennek az az oka, hogy a CPU esetén sokkal nagyobb jelentősége van magának az algoritmusnak, mint a GPU esetén, mivel a legtöbb (nagyon) összetett algoritmus nem implementálható hatékonyan GPU segítségével.



2. ábra. GPU alapú mátrix szorzó algoritmusok futásideje

A CPU és a GPU alapú algoritmusokat összehasonlítva (3. ábra) azt látjuk, hogy nagy mátrixok szorzása esetén még a legegyszerűbb GPU alapú algoritmus is jelentősen gyorsabb mint a nagyon erősen optimalizált CPU-s algoritmus (GNU GSL CBLAS). Az erősen optimalizált CUBLAS implementáció pedig körülbelül 3 és fél nagyságrenddel gyorsabb mint, a leggyorsabb CPU-s verzió. Viszont még az itt tapasztalt sebességnövekedés sem éri el a CPU és a GPU közötti elméleti teljesítménykülönbséget. A nagy sebességkülönbségnek az oka, hogy a nagy mátrixok szorzása az egyik leghatékonyabban megvalósítható algoritmus CUDA segítségével. Két $n * n$-es mátrix összeszorzásához összesen $O(n^2)$ adat mozgatására van szükség. Ez nagy n-ek esetén jelentősen kevesebb, mint az elvégzendő műveletek száma, ami a használt algoritmustól függően általában $O(n^3)$ és $O(n^{2.81})$ (Strassen algoritmus [14]) között van. A jelenleg ismert algoritmusok, amelyek ennél aszimptotikusan hatékonyabbak, a gyakorlatban nem teljesítenek jól (a túlságosan nagy konstans miatt). Szintén a hatékony CUDA-s implementálhatóságot javítja az, hogy mind a legegyszerűbb $O(n^3)$-ös, mind a Strassen algoritmus csak nagyon egyszerű műveleteket használ (összeadást, kivonást és szorzást).



3. ábra. CPU és GPU alapú mátrix szorzó algoritmusok futásideje

2.3. A GPU alapú algoritmusok sebességének összehasonlítása

Az egy szálon futó algoritmusok sebessége jól jellemezhető az algoritmus szükséges lépésszámával, mivel egyszerre mindig egy műveletet hajt végre a processzor. Ezzel szemben a GPU esetén a szükséges műveletek száma csak részben tudja jellemezni egy algoritmus (elméleti) hatékonyságát. GPU-k esetén a szükséges lépések száma mellett még egy mérőszámra van szükség ahhoz, hogy egy algoritmus hatékonyságát lehessen jellemezni. Ez a paraméter az iterációk száma (step complexity), ami azt mutatja meg, hogy hány lépésre lenne szükség az algoritmus végrehajtásához abban az esetben, ha tetszőleges darabszámú processzor állna rendelkezésre, tehát minden olyan műveletet, amelyik elméletileg egyszerre végrehajtható, azt a rendszer ténylegesen egyidőben végre is hajtja.

Általában igaz, hogy kis méretű problémák esetén az iterációk számával, nagy problémák esetén pedig a szükséges lépések számával lehet jobban közelíteni a valós futásidő skálázódását. Ennek az az oka, hogy kis problémák esetén elég jó közelítéssel teljesül az, hogy az összes párhuzamosan elvégezhető műveletet el tudja végezni a GPU egyszerre, míg nagy problémák esetén a GPU-ra fejlesztett algoritmusok kevésbé jól párhuzamosítható részei is ki tudják használni a GPU-ban lévő összes processzor számítási kapacitását.

Párhuzamos redukciónál például a szükséges lépések száma $O(n)$, ami megegyezik a soros implementáció lépésszámával, viszont a szükséges iterációk száma csak $O(log(n))$. Az első iteráció során viszont összesen $\frac{n}{2}$ műveletet kell végrehajtani, amit már nem túl nagy n-ek esetén sem tud egyszerre elvégezni a videokártya. Ezért kis n-ek esetén a videokártyás implementáció futásideje $O(log(n))$ szerint skálázódik, majd n növekedésével a skálázódás $O(n)$-essé változik.

3. Forgalmi mátrix számítása linkterhelésekből

Egy IP hálózat forgalmi mátrixa megadja, hogy a hálózat egyes végpontjai között mennyi adat áramlik. Ez az információ jól használható a hálózat fejlesztéséhez, de a közvetlen megmérésére széleskörűen alkalmazható gyakorlati megoldás nem áll rendelkezésre. Ezért a forgalmi mátrixot az egyszerűen megmérhető linkterhelések alapján és a hálózat állapotáról rendelkezésre álló egyéb adatok (mint például a pont-pont kommunikációkhoz használt utak) alapján szokás becsülni.

3.1. Jelenleg ismert és használt megoldások

3.1.1. Gravitációs modellen alapuló megoldások

A legegyszerűbb és legrégebb óta létező megoldások a forgalmi mátrix meghatározására a gravitációs modelleken alapulnak ([8, 9, 13]). Ezeknek a megoldásoknak előnye, hogy nagyon kis számítási kapacitást igényelnek, viszont nagyon pontatlanok, ezért manapság már nem használják őket.

Ezek a megoldások a következő egyenlőségre alapulnak:

$$X_{i,j} = \frac{R_i \cdot A_j}{f_{i,j}}$$

ahol $X_{i,j}$ a hálózat $i.$ pontjából a hálózat $j.$ pontjába tartó forgalom mennyisége, R_i a hálózatba az $i.$ pontban belépő, A_j pedig a $j.$ pontban kilépő forgalom mennyisége, $f_{i,j}$ pedig az arány mátrix megfelelő eleme. A_j és R_j egyszerűen megmérhető, de az egyenlet megoldásához szükséges f mátrix pontos meghatározása egy ugyanolyan bonyolultságú feladat, mint az X mátrix meghatározása. Ezt a gravitációs modellek úgy oldják meg, hogy az f mátrix elemeit valamilyen nagyon egyszerű algoritmussal becsülik (például a mátrix összes eleme 1) és ez alapján számolják ki az X mátrix elemeit.

3.1.2. Hálózati tomográfián alapuló megoldások

A hálózati tomográfián alapuló forgalmi mátrix becslő algoritmusok ([2, 10, 15]) abból indulnak ki, hogy a forgalmi mátrix elemei valamilyen konkrét valószínűségi eloszlást követnek. Ezek az algoritmusok feltételeznek egy (általában nevezetes) eloszlást, aminek a paraméterei a megmért linkterhelésekből kinyert adatok. Ezen adatokat és a feltételezett valószínűségi eloszlást felhasználva adnak egy becslést a forgalmi mátrixra. Bizonyos algoritmusok ezt a becslést a linkterhelések és a hálózatban szereplő utak együttes felhasználásával pontosítják.

A gravitációs modellnek és a hálózati tomográfiának az együttes használatával a többi algoritmushoz képest kifejezetten jó eredményeket értek el 2003-ban [16]. Az amerikai gerinchálózatból származó adatokra alkalmazva az algoritmusokat, sikerült elérniük, hogy a forgalmi mátrix elemeinek 90%-ának relatív hibája kisebb lett mint 23% úgy, hogy a relatív nagy hibákat a kis értékű (ezért kevésbé jelentős) elemeknél követték el.

3.1.3. Neurális hálózatokon alapuló megoldások

Az utóbbi pár évben elkezdték a neurális hálózatokon alapuló megoldások használatát is a forgalmi mátrixok kiszámításához ([3, 6, 7]). Ezeknek a megoldásoknak az a hátránya, hogy nagy mennyiségű adatra van szükség a betanításukhoz, ami általában nem áll rendelkezésre. Amennyiben viszont valamilyen adatokkal megfelelően be tudtuk tanítani őket, akkor az összes többi megoldásnál nagyobb pontosságot tudnak elérni. Mivel a neurális hálózatok egy jelenleg is aktív kutatási téma, ezért az ott elért eredményekkel párhuzamosan várhatóak új megoldások a forgalmi mátrix becslésének neurális hálózatokon alapuló megközelítésében is.

3.2. A probléma matematikai modellezése

A probléma hatékony számítógépes kezeléséhez fel kell állítani egy matematikai modellt, amelyik jól leírja a mérnöki probléma egyes paramétereit, viszont elég egyszerű ahhoz, hogy megoldható legyen. Én a probléma megoldásához felhasználom a hálózatot reprezentáló gráf struktúráját az egyes végpontok közötti kommunikációhoz használt útvonalakkal együtt (két végpont között több különböző útvonal az ECMP (Equal Cost Multi-Path) miatt lehetséges). Ezen kívül felhasználom a hálózat egyes összeköttetésein (irányított élein) átáramló adat mennyiségét, ami a konkrét mért adat. Ezen kívül felhasználok egy becslést a forgalmi mátrixra, amelyik a hálózat minden egyes végpont párjára tartalmaz egy becslést az azon két pont között áramló forgalomra. Ez a becslés a hálózat egyes végpontjaiban szereplő felhasználók ismeretén (például adat központ, web szerver, lakossági felhasználók, ...) alapul.

A matematikai modellezés célja, az adatok egyenletek, egyenlőtlenségek és függvények segítségével történő leírása azért, hogy utána hatékonyan fel lehessen dolgozni számítógép segítségével. A hálózat jól modellezhető egy irányított gráffal, ahol a gráf pontjai a hálózat csomópontjainak felelnek meg, a gráf élei pedig a hálózatban szereplő összeköttetéseket reprezentálják. A hálózatban szereplő kétirányú összeköttetéseket két, ellentétes irányú, irányított éllel reprezentálom. A gráf minden éléhez és minden csomópontjához hozzárendelek 1-1 indexet, és a továbbiakban az i. élre l_i-ként, az i. pontra pedig p_i-ként fogok hivatkozni. A továbbiakban n fogja jelölni a gráf pontjainak a számát, m pedig a gráf éleinek a számát.

A meghatározandó forgalmi mátrixot jelölje X, ahol X egy n^2 elemű oszlopvektor, ahol a vektor i. eleme (0-tól kezdődő indexelést használva) a $\lfloor \frac{i}{n} \rfloor$ indexű csomópontból a $i \bmod n$ indexű csomópontba menő forgalmat jelenti (4. ábra). Az oszlopvektor használatára azért van szükség, mivel így a továbbiakban elkerülhető lesz a 3 dimenziós mátrixok használata, amelyek matematikai kezelése viszonylag bonyolult. Számítástechnika szempontból pedig minden mátrix 1 dimenzióban van tárolva a memóriában, ezért ez a döntés ott sem okoz problémát.

A hálózat és a meghatározandó adat modellezése után ehhez a modellhez hozzá kell illeszteni a rendelkezésre álló mért adatokat is. A mért linkterhelések jól reprezentálhatóak egy L oszlopvektorral, ami m elemet tartalmaz, ahol a vektor i. eleme az l_i él terhelését jelenti. Az egyes végpontok között használt utak reprezentációjához pedig egy R mátrixot használok, amelyiknek m sora és n^2oszlopa van. A mátrix minden egyes oszlopa a hálózat adott két végpontját összekötő utakat reprezentálja

úgy, hogy a j. oszlop a $\lfloor \frac{j}{n} \rfloor$ indexű csomópontból a $j \bmod n$ indexű csomópontba vezető utak adatait tartalmazza. Egy oszlop i. eleme megmutatja, hogy az adott oszlop által reprezentált csomópontpár közötti kommunikáció hány százaléka halad át az l_i élen. A mátrixok ilyen módon történő megválasztása esetén a mért linkterhelésekből és a végpontok közötti utakból származó feltétel megfogalmazható a következő egyenlettel:

$$R \cdot X = L$$

A linkterheléseken kívül még be kell építeni a modellbe a forgalmi mátrix becslését, amivel a probléma megoldását egyértelműsíteni lehet, mivel az előző egyenletrendszer erősen alul határozott. Ennek az az oka, hogy egy hálózatban mindig sokkal kevesebb él van, mint ahány csomópont pár (a valós hálózatok soha sem felelnek meg egy teljes gráfnak). A forgalmi mátrix becslését tartalmazó mátrixot jelölje E. Ennek a mátrixnak a felépítése egyezzen meg az X mátrix szerkezetével. Ezen kívül vegyünk fel egy f függvényt, ahol $f : \mathbb{R}^{n^2 \times 1} \times \mathbb{R}^{n^2 \times 1} \to \mathbb{R}$. Ez a függvény határozza meg, hogy a számított és a becsült mátrix mennyire hasonlít egymásra úgy, hogy a függvény értéke csökken, ha a két mátrix hasonlósága nő. Két mátrix hasonlóságát többféleképpen is lehet definiálni, ezért a forgalmi mátrix meghatározásánál több különböző f függvényt is meg fogok vizsgálni. A probléma megoldása során a cél, hogy ennek az f függvénynek az értékét minimalizáljam úgy, hogy közben az X mátrix kielégíti a linkterhelésekből adódó mátrix egyenletet is.

0.	1.	...	n-2.	n-1.
n.	n+1.	...	2n-2.	2n-1.
...	...	...	...	...
n^2-n.	n^2-n+1.	...	n^2-2.	n^2-1.

4. ábra. Az X vektor indexeinek jelentése 2D-s mátrixban

Ez a matematikai modell arra a feltevésre épít, hogy a rendelkezésre álló forgalmi mátrix becslésünk gyakorlati szempontból elfogadható pontosságú, viszont nem konzisztens a mért linkterhelésekkel. A modell által specifikált probléma megoldásával ezt az inkonzisztenciát lehet feloldani ami azért lényeges, mivel a forgalmi mátrix felhasználása során problémát jelenthet az, ha a benne szereplő adatok ellentmondanak az egyéb rendelkezésre álló adatokkal.

4. Lineáris programozás

A lineáris programozás egy elterjedt technika olyan optimalizálási feladatok megoldására, ahol egy célfüggvénynek a maximumát vagy a minimumát szeretnénk meghatározni valamilyen feltételek kielégítése mellett. A lineáris programozás abban az esetben használható, ha mind a célfüggvény, mind a feltételek felírhatóak a döntési változók lineáris függvényeként. Amennyiben a döntési változók tetszőleges valós értéket felvehetnek (amelyek megfelelnek a lineáris feltételeknek), akkor a problémára létezik legrosszabb esetben is polinomiális idejű megoldás. Ugyanakkor a gyakorlatban a szimplex algoritmust szokás használni, melynek a futásideje bizonyos esetekben exponenciális, de a gyakorlati problémák esetén gyorsabb, mint a polinomiális algoritmusok. Amennyiben a döntési változók csak egész értékeket vehetnek fel (egész értékű probléma), akkor az optimalizálási probléma már NP teljes, tehát jelen ismereteink szerint nem oldható meg hatékonyan. Az egész értékű problémákkal a továbbiakban nem fogok foglalkozni.

4.1. A szimplex algoritmus

A szimplex algoritmus a nevét onnan kapta, hogy geometriailag leírható úgy, hogy egy n dimenziós komplex poliéder (szimplex) csúcsain ugrálunk és ezzel keressük az optimális megoldást. Ugyanez az algoritmus leírható lineáris algebra segítségével is. Mivel ez a számítógépek számára egy sokkal kedvezőbb reprezentáció, ezért a továbbiakban ezt fogom használni.

Minden lineáris programozási probléma leírható a következő alakban:

$$\max c^T \cdot x$$

$$A \cdot x \geq b$$

$$x \geq 0$$

Ez az alak továbbá átírható úgy, hogy bevezetem az x_s segédváltozókat és ezzel az eredeti feltételeket egyenlőségekként írom fel.

$$\max c^T \cdot x$$

$$A \cdot x - x_s = b$$

$$x \geq 0$$

$$x_s \geq 0$$

Ezt az alakot szokás a probléma sztenderd alakjának hívni. A továbbiakban ezt az alakot fogom használni. Továbbá n jelöli az eredeti problémában szereplő döntési változók számát, m pedig az eredeti problémában szereplő feltételek számát (a döntési változók nem negativitását kikötő fenti feltételek nélkül).

A szimplex algoritmus egy két fázisból álló iteratív algoritmus. Az első fázisban az algoritmus a problémának egy olyan megoldását keresi meg, amelyik az összes feltételt kielégíti, de nem optimális. Ezt követően a második fázis az első fázis eredményéből kiindulva keresi meg a probléma optimális megoldását. A két fázis

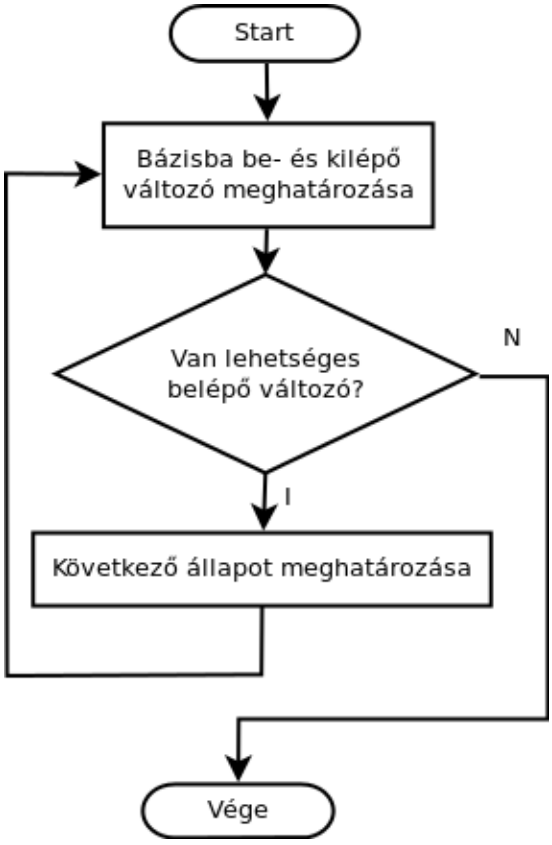
megvalósítását tekintve gyakorlatilag megegyezik (ugyanazt az algoritmust kell végrehajtani két különböző adaton), ezért a továbbiakban csak a második fázissal fogok foglalkozni. Az algoritmus egy állapotát az

$$x_B = b + A \cdot x_{NB}$$

$$z = c_0 + c^T \cdot x_{NB}$$

egyenletekkel lehet leírni, ahol x_B a bázist alkotó változókat tartalmazó vektor, az x_{NB} pedig a bázisban nem szereplő változókat tartalmazó vektor. A két vektorban szereplő változók diszjunktak, és az x és az x_s vektorok összes eleme szerepel az egyik vektorban.

Az iteratív algoritmus minden iterációja három lépést tartalmaz (5. ábra). Az első lépésben meghatározza, hogy melyik változó lép be a bázisba és melyik változó lép ki a bázisból. Amennyiben nem talált olyan változópárt, amelyek lehetnének be- és kilépő változók, akkor kilép az iteratív algoritmus, mivel ebben az esetben az algoritmus megtalálta az optimális megoldást vagy arra az eredményre jutott, hogy a problémának nem létezik optimális megoldása (az optimalizálandó függvény a megadott feltételek mellett felülről nem korlátos). Amennyiben viszont talált olyan be- és kilépő változót, amely megfelel a feltételeknek, akkor az iteráció utolsó lépésében egyszerű lineáris algebrai műveletek segítségével megváltoztatja az A, a b és a c mátrixokat és megváltoztatja c_0 értékét úgy, hogy az egyenletek jelentése ne változzon meg. Tehát ugyanazok az x értékek elégítsék ki az átalakítás utáni egyenleteket, mint amik az átalakítás előtti egyenleteket is kielégítették. Az egyenletrendszer egy megoldása, ha a bázisban nem szereplő változók értékét 0-nak vesszük. Ekkor a célfüggvény (z) értéke meg fog egyezni c_0 értékével. Az algoritmus előrehaladása során ez az érték soha sem csökken.



5. ábra. A szimplex algoritmus folyamatábrája

4.2. A szimplex algoritmus CPU-s implementációja

Jelenleg nagyon sok nyílt forráskódú, illetve fizetős megoldó szoftver érhető el lineáris programozási problémák megoldására. Ezek közül gyakorlatilag az összes a szimplex algoritmust használja, esetenként egyéb optimalizációkkal továbbfejlesztve. A fejlettebb szoftverek például a szimplex algoritmus lefuttatása előtt végrehajtanak egy előfeldolgozást, amely általában le tudja csökkenti a problémában szereplő

változók és feltételek számát úgy, hogy az eredeti probléma ne változzon meg. Mivel a dolgozatomnak nem témája ezeknek az előfeldolgozó algoritmusoknak az elemzése és implementálása, ezért elkészítettem a szimplex algoritmus CPU-s referencia implementációját. Ennél az implementációnál minden iterációban a legnagyobb együtthatójú belépő változót választom ki a bázisba belépő változónak.

A referencia implementációra azért van szükség, hogy a GPU alapú implementációmat egy hasonló bonyolultságú és hasonló szinten optimalizált CPU-s implementációval is össze tudjam hasonlítani. Az előfeldolgozás kihagyása nem jelent számottevő különbséget a CPU és a GPU alapú implementációk futásidejének arányában, mivel mind a két esetben körülbelül ugyanakkora javulást jelentene a futásidőben (az előfeldolgozó futásideje általában elhanyagolható a szimplex algoritmus futásidejéhez képest).

4.3. A szimplex algoritmus GPU-s implementációja

A következőkben megvizsgálom, hogy a szimplex algoritmus egyes lépései (optimalitás vizsgálat, be- és kilépő változó meghatározása, következő állapot meghatározása) hogyan valósíthatóak meg GPU segítségével. Minden lépésre bemutatok több különböző implementációt, és összehasonlítom őket futásidő és egyéb felhasználhatósági szempontok alapján.

4.3.1. A bázisba be- és kilépő változók meghatározása

A legtöbb algoritmus, amelyik a bázisba be, illetve onnan kilépő változót határozza meg, csak $O(n)$ lépést igényel, ami nagy problémák esetén is kevés időt vesz igénybe. Ezek az algoritmusok a CPU esetén igen kedvezőek, viszont a GPU esetén nem használják ki a teljes számítási kapacitást. GPU használata esetén érdemes olyan algoritmusokat is megvizsgálni, amelyek jelentősen nagyobb számítási kapacitást igényelnek, mivel a számítási igény növekedésénél lassabban fog nőni a szükséges futásidő. Ezzel egy, az optimális megoldáshoz viszonylag gyorsan konvergáló, viszont CPU esetén túlságosan időigényes algoritmus használata GPU esetén kifejezetten kedvező eredményeket nyújthat.

A különböző algoritmusok tesztelésénél olyan problémákat vizsgálok, ahol a problémában szereplő konstansok egyenletes eloszlású véletlen számok, és kétszer annyi feltétel szerepel a problémában, mint ahány ismeretlen. Az algoritmusok összehasonlításához az optimális megoldáshoz történő konvergencia sebességét és az egy ki- és belépő változó pár meghatározásához szükséges időt veszem alapul.

Legnagyobb együtthatójú belépő változó választása. Ezt a heurisztikát a szimplex algoritmus kitalálója, George Dantzig javasolta. Az algoritmus lényege, hogy mindig azt a változót választja ki a belépő változónak, amelynek a legnagyobb az együtthatója a célfüggvényben. Több azonos együtthatójú változó esetén véletlenszerűen választ. Ez az algoritmus $O(n)$ lépést igényel és GPU esetén $O(log(n))$ iterációval megoldható. A megoldás egy egyszerű maximumredukció segítségével meghatározható annyi módosítással, hogy a maximum értéke helyett itt a maximum helyét kell meghatározni, ami viszont nem okoz további nehézségeket. A belépő változó meghatározása után egy második redukció segítségével a kilépő változó

meghatározása $O(m)$ lépésben és összesen $O(log(m))$ iterációval megoldható. Tehát az algoritmus teljes futásideje $O(n+m)$ lépés és összesen $O(log(n)+log(m))$ iteráció.

Legnagyobb előrelépést biztosító belépő változó választása. Ezt a heurisztikát szokás mohó heurisztikának is hívni, mivel mindig azt a változót választja a bázisba belépő változónak, amelyik a jelenlegi iterációban a legjobban megnöveli a célfüggvény értékét. Ez úgy valósítható meg, hogy minden egyes lehetséges belépő változó (ahol a változó együtthatója a célfüggvényben pozitív) esetén meghatározza, hogy annak a választása esetén mennyivel növekedne a célfüggvény, majd azt választja, amelyik esetén ez az érték a legnagyobb.

Minden egyes lehetséges belépő változóhoz meg kell határozni a célfüggvény megváltozását és ezzel egyidőben a hozzátartozó kilépő változót is. Ez összesen $O(n \cdot m)$ lépésben oldható meg, viszont optimális implementáció esetén mindössze $O(log(m))$ iterációra van hozzá szükség. Ezt követően a legnagyobb előrelépést biztosító belépő változó meghatározása már csak $O(n)$ lépés és $O(log(n))$ iteráció. Ezeket összeadva megkapjuk, hogy ennél az algoritmusnál a szükséges lépések száma $O(n \cdot m)$ és a szükséges iterációk száma pedig $O(log(n) + log(m))$. Tehát ez az algoritmus sokkal több műveletet hajt végre, viszont elméletben (ha a szálak számának növekedése nem lassítaná az egyes szálak futásidejét) közel ugyanolyan gyorsan le tud futni, mint a legnagyobb együtthatójú belépő változót választó algoritmus. Mivel ennek az algoritmusnak az első része (minden lehetséges változóra a célfüggvény megváltozásának meghatározása) bizonyos körülmények között képes kihasználni a GPU számítási kapacitásának jelentős részét, ezért megvizsgálok 4 különböző implementációt is az optimális megoldás kiválasztásához.

Belépő változónként egy szál használata. A legegyszerűbb megoldás, hogy minden egyes belépő változóhoz tartozik egy szál, amelyik meghatározza az elérhető növekményt az adott változónak a kiválasztása esetén. Ennek az implementációnak a fő problémája, hogy minden egyes szálnak m elem közül kell kiválasztani a legkisebbet, amihez $O(m)$ iterációra van szükség és mindössze n szál fut egyszerre. Ez a mennyiség még viszonylag nagy problémák esetén is kevés ahhoz, hogy a GPU teljes számítási kapacitását kihasználja.

Belépő változónként egy blokk használata. Az előző algoritmus problémájára egy egyszerű megoldás, hogy minden egyes lehetséges belépő változóhoz egy szál helyett egy egész blokk tartozik. Ekkor a blokk be tudja tölteni az adatokat a blokk szinten megosztott (shared) memóriába és utána ott hatékonyan végre lehet hajtani a legkisebb elem megkeresését egy redukció segítségével. Ez az algoritmus már képes lenne kihasználni a videokártya teljes számítási kapacitását, viszont az adatok betöltése a blokk szintű memóriába nem hatékony. Ennek az az oka, hogy az egyszerre futó (egy warp-ban levő) szálak az adatokat tartalmazó tömb egy oszlopát olvassák be, aminek az elemei a memóriában egymástól messze helyezkednek el a C nyelv tömb kezelése miatt (sor alapú elrendezés). Így a memória sávszélességének csak kis részét képes kihasználni ez az algoritmus, aminek következtében a memóriaolvasások sebessége a futásidőt limitáló tényezővé válik. Az adatokat tar-

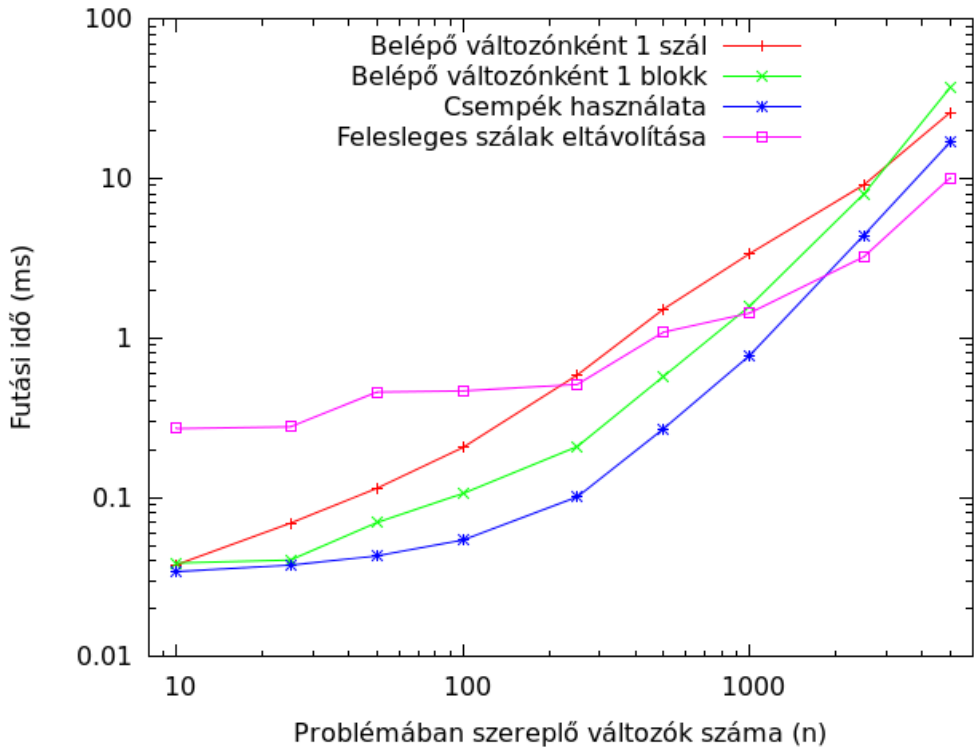
talmazó tömb transzponáltjának tárolása megoldást jelentene a problémára, viszont ez a későbbiekben komoly problémákat okozna.

Csempék használata a memória hozzáférések javításához. Az előző algoritmusban szereplő nem optimális memóriaolvasási mintának a megoldására egy elterjedt megoldás a csempék használata. Jelen esetben a csempék használatát úgy célszerű megoldani, hogy egy blokk összesen 16 változóhoz számolja ki, hogy annak a választása esetén mekkora növekedés érhető el a célfüggvényben. Ekkor egy lehetséges belépő változó esetén a célfüggvényben elért növekedést a blokkméret / 16 szál számolja ki, ami a jelenlegi GPU-k esetén elemenként 32 vagy 64 szálat jelent. Ezzel a módosítással a GPU számítási kapacitásának kihasználtságát a két előző algoritmus közötti szinten lehet biztosítani - ami a gyakorlatban jó kihasználtságot jelent, viszont a második algoritmus futásidejét növelő tényezőt, a globális memória hozzáférések idejét le lehet csökkenteni. Ennek az az oka, hogy 16 egymás utáni szál (fél warp) az adatokat tartalmazó tömb 16 darab egymás utáni elemét olvassa ki, ami 8 byte-os (például double) elemekkel számolva 128 byte. Ez azért optimális, mivel a GPU-k globális memóriája úgy működik, hogy minden esetben 128 byte-ot olvas ki és amennyiben nincsen szükség az összes adatra, akkor a feleslegesen kiolvasott adatot eldobja.

Valós munkát nem végző szálak eltávolítása. A csempék használatával már egy elég jó eredményt sikerült elérni, viszont abban az implementációban a szálaknak jelentős része semmilyen érdemi munkát sem végez. Ennek az az oka, hogy amelyik változónak a célfüggvényben szereplő együtthatója negatív, az biztosan nem léphet be a bázisba, ezért azoknál nincsen arra szükség, hogy kiszámoljam az elérhető előrelépést. Ennek ellenére az előző implementáció ezekhez a változókhoz is hozzárendeli a megfelelő darabszámú szálat, amik nem csinálnak semmit, viszont ennek ellenére ugyanannyi ideig foglalják le a GPU egy processzorát, mivel nagy valószínűséggel egy futtatási egységben (warp) lesznek legalább egy olyan szállal, amelyik tényleges munkát is végez.

Ez a probléma úgy oldható meg, hogy először minden változóra megvizsgálom, hogy beléphet-e a bázisba, és ennek alapján a következő kernelt úgy indítom el, hogy csak azokhoz a változókhoz tartozzon szál, amelyek célfüggvényben lévő együtthatója pozitív (beléphet a bázisba). Ennek a megvalósításához egy olyan tömböt kell létrehozni, amelyiknek az első k elemében vannak azon elemek indexei, amelyek beléphetnek a bázisba. Ezt a legegyszerűbben egy particionálás segítségével lehet megvalósítani, ahol a particionálás feltétele az, hogy az adott indexű elem beléphet-e a bázisba. A particionálás megvalósításánál a Thrust [5] osztálykönyvtárban lévő stabil particionáló algoritmust használtam. A stabilitásra azért volt szükség, hogy az egymáshoz közeli memória címeket olvasó szálak egymás mellett maradjanak. A particionáláshoz összesen $O(n \cdot log(n))$ lépésre és $O(log(n))$ iterációra van szükség.

A 4 implementáció összehasonlítása. Mivel a négy implementáció csak a GPU-s kernel megvalósításában különbözik egymástól, ezért a szimplex algoritmus mind a négy esetben pontosan ugyanolyan gyorsan fog konvergálni az optimális



6. ábra. A legnagyobb előrelépést biztosító belépő változó meghatározását végző implementációk összehasonlítása

megoldáshoz. Ezt kihasználva, az összehasonlításukhoz elég az egyes implementációk futásidejét vizsgálni. A négy implementáció futásideje az 6. ábrán látszik a probléma méretének a függvényében.

Az ábráról látszik, hogy az első algoritmus kis n-ek esetén lassabb, mint a második és a harmadik, viszont nagy n értékek esetén már a második implementációnál jobban teljesít. Ennek az az oka, hogy nagy problémák esetén már az első implementáció is elég sok szálat használ ahhoz, hogy jobban teljesítsen, mint a második implementáció, amelyik a memória sávszélességét nagyon rosszul használja ki.

A második és a harmadik implementációt összehasonlítva egyértelműen látszik, hogy milyen sokat számít a memória sávszélességének hatékony kihasználása. A hatékonyabb memória hozzáférési mintát használó algoritmus körülbelül feleannyi idő alatt lefut, mint a másik, annak ellenére, hogy kevesebb szálat használ.

A negyedik implementáció futásideje kicsit máshogy skálázódik, mint az előző két implementáció futásideje. Kis problémák esetén sokkal lassabban fut le, mint a többi algoritmus, mivel a particionáláshoz relatív sok időre van szükség a teljes futásidőhöz képest. Ennek az az oka, hogy a Thrust osztálykönyvtár elsősorban nagy adatmennyiségek kezelésére lett optimalizálva. Nagy problémák esetén viszont már egyértelműen megéri kiszűrni azokat a változókat, amelyek nem léphetnek be a bázisba és ezzel optimalizálni a szükséges számítási kapacitás mennyiségét. A legnagyobb vizsgált problémák esetén ezzel a megoldással körülbelül 1.6-szor olyan gyorsan futott le a ki- és belépő változót kiválasztó algoritmus, mint az optimalizáció nélkül.

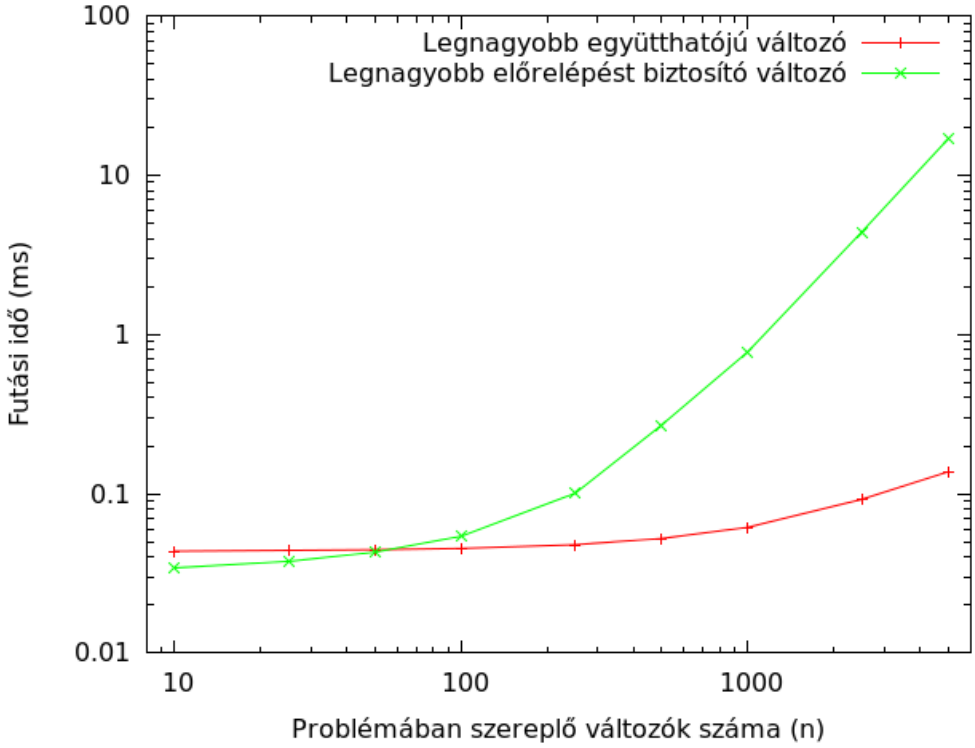
Az implementációkat Nvidia Visual Profiler segítségével vizsgálva jól látszanak a különböző implementációk gyenge pontjai. Az első implementációnál a legkritikusabb pont a processzorok terhelése a kevés számú szál miatt, viszont ez a probléma a bemenő adat méretének növelésével csökken. Elég nagy méretű bemenő adat esetén sebessége elérhetné a harmadik implementáció sebességét is, viszont ehhez olyan nagy mennyiségű bemenő adatra lenne szükség, ami már nem férne el a videokártya memóriájában.

A második implementáció esetén a legkritikusabb limitáló tényező a memóriaolvasások sebessége. A méréseim szerint a második implementáció a beolvasott adat mennyiségnek mindössze a 6%-át használja fel, ami nagyon erősen lelassítja a futási sebességet. Ugyanez a szám a többi implementációnál 25-30% körül mozog, ami szintén alacsony, viszont a második implementációnál mérhető 6%-nál sokkal jobb. A második implementációnál az alacsony értéket a szálak nem megfelelő elrendezése okozza. A többi implementációnál pedig azért ilyen alacsony ez a szám, mivel a memória működéséből következően azokhoz a változókhoz tartozó adatokat is beolvassa, amelyek nem lehetnek belépő változók és azokat az adatokat egyik szál sem használja fel.

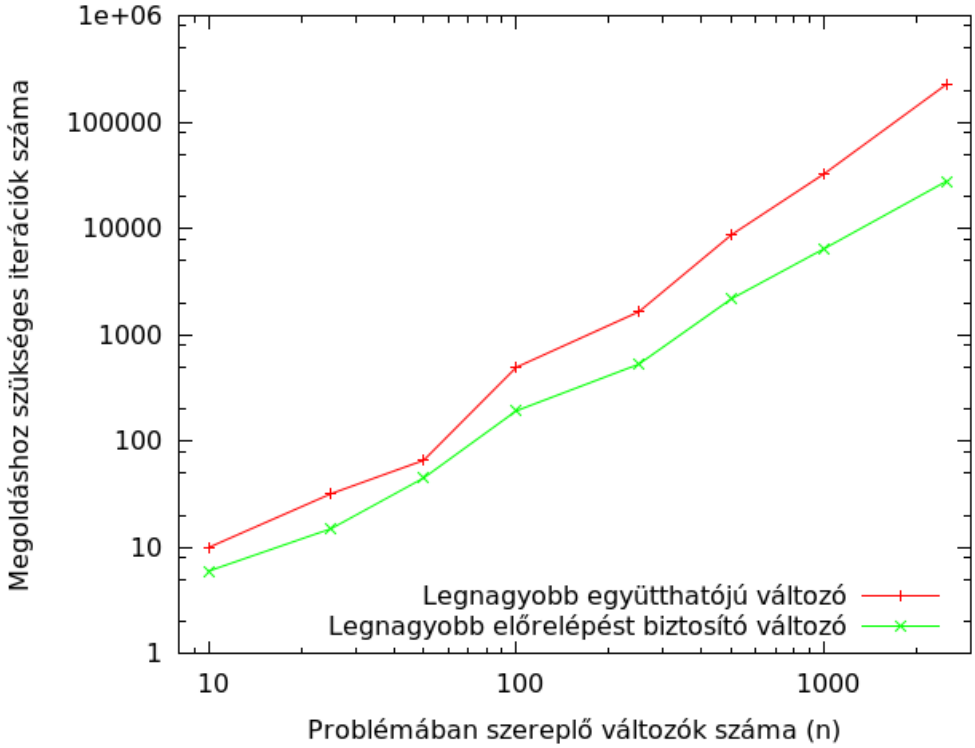
A negyedik implementáció által elért sebességnövekedést az okozza, hogy a valós munkát végző szálakat egymás utánra csoportosítottam. Ez a hatás a profilozás során is egyértelműen látszik, mivel a második és a harmadik implementációnál a végrehajtás hatékonysága 30% körül mozgott, a negyedik implementációnál viszont már közel 65% volt.

A végrehajtás hatékonysága azt jelenti, hogy az elvégzett műveletek hány százaléka volt ténylegesen hasznos. Az alacsony végrehajtási hatékonyságot az okozza, hogy minden egyes szál elvégzi az összes szükséges műveletet, és utána amelyiknek nem kellett volna elvégeznie egy adott műveletet, az nem tárolja el az eredményt a megfelelő helyre. A legtipikusabb probléma, ami alacsony végrehajtási hatékonyságot eredményez, az a "warp divergency" néven ismert jelenség. Ez azt jelenti, hogy a közvetlenül egymás mellett lévő szálak nem ugyanazokat az utasításokat hajtják végre.

A különböző algoritmusok összehasonlítása. A ki- és belépő változó meghatározására bemutatott két különböző algoritmus összehasonlítása látható a 7. és a 8. ábrán. Egyértelműen látszik, hogy a legnagyobb előrelépést biztosító változó választása a vizsgált problémák esetén gyorsabb konvergenciát biztosít, viszont sokkal több időre van szükség a ki- és belépő változók meghatározásához. A két különbség együttes hatása azt eredményezi, hogy a CPU-s esettel ellentétben itt a mohó algoritmus teljesít jobban, mivel a gyorsabb konvergenciának köszönhetően a következő állapot meghatározását is kevesebbszer kell végrehajtani. Ennek az az oka, hogy a GPU komoly számítási kapacitását a mohó algoritmus jobban ki tudja használni, ezért relatív kevesebb időt veszt el a GPU a bonyolultabb ki- és belépő választás miatt, mint amennyit a CPU.



7. ábra. A két algoritmus futásidejének összehasonlítása (a legnagyobb előrelépést elérő változó kiválasztásánál a csempéket használó algoritmus)



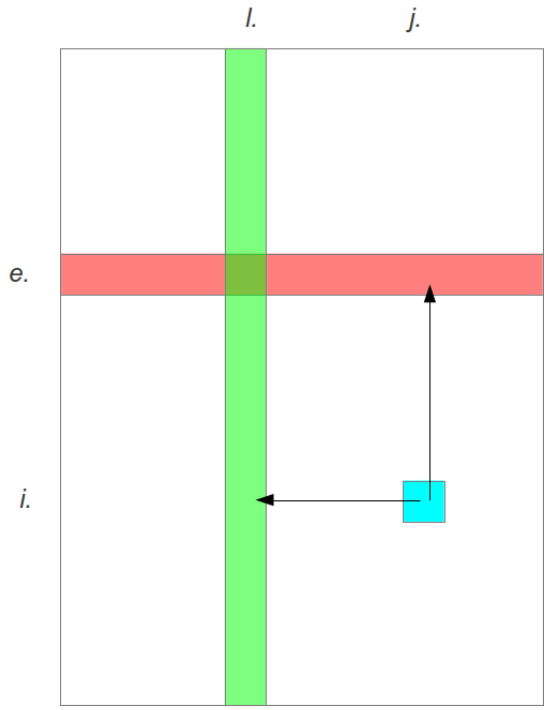
8. ábra. A két algoritmus konvergenciájának összehasonlítása

4.3.2. A következő állapot meghatározása (pivoting)

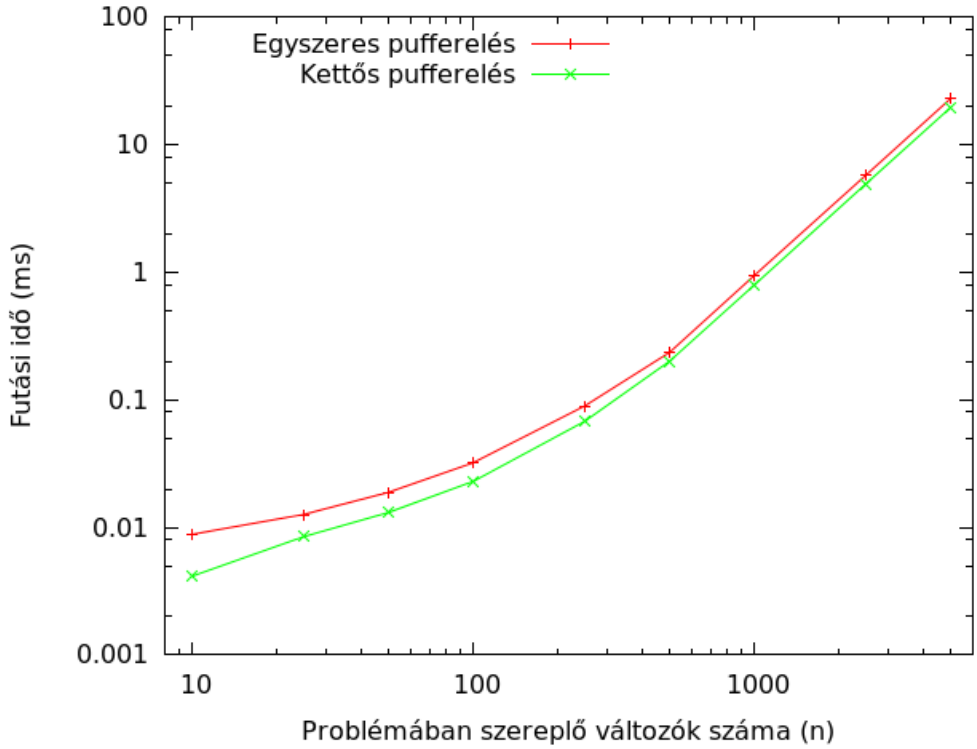
A szimplex algoritmus legfontosabb része a következő állapot meghatározása. Általában ez az algoritmus legidőigényesebb része, ezért ennek az implementálására komoly figyelmet kell fordítani.

Kettős pufferelés használata. A kettős pufferelés lényege, hogy összesen két puffert használva a szimplex algoritmus minden iterációjánál az egyik pufferből olvasom ki a jelenlegi állapotot és a másik pufferbe írom bele a következő állapotot, majd megfordítom a két puffer szerepét. Ezt kihasználva a következő állapot kiszámítása során az nem okozhat problémát, hogy egy elemet korábban módosított le az egyik szál, mint amikor egy másik szál ugyanarról a helyről még az előző értéket próbálta kiolvasni. Így elég egy kernel ahhoz, hogy meghatározzam a szimplex algoritmus következő állapotát. A kettős pufferelés hátránya, hogy a megvalósításához összesen kétszer akkora területet kell lefoglalni a memóriában, mint amire a szimplex algoritmus egy állapotának az eltárolásához szükség van.

Egyszeres pufferelés használata. Amennyiben nincsen elég memória a GPU-ban ahhoz, hogy a szimplex állapotának tárolásához feltétlenül szükséges memóriamennyiség kétszeresét foglaljuk le, akkor egyszeres pufferelés használatára van szükség. Ez az eset könnyen bekövetkezhet, mivel a szimplex algoritmus GPU-s implementációjának skálázódását legkomolyabb a GPU-ban elérhető memória mennyisége (általában 1GB) limitálja. Az egyszeres pufferelés megvalósításához fontos, hogy az állapotot reprezentáló tömb egyes elemeinek a kiszámításához melyik egyéb elemekre van szükség. Legyen a bázisba belépő változó az $e.$ sorban, a bázisból kilépő változó pedig az $l.$ oszlopban. Ekkor az $i.$ sorban és a $j.$ oszlopban lévő elem az $e.$ sor $j.$ elemétől és az $i.$ sor $l.$ elemétől függ (9. ábra). Ez alapján először azokat az elemeket kell kiszámolni, amelyek sem az $e.$ sorban, sem az $l.$ oszlopban nincsenek benne. Ezt követően ki lehet számolni az $e.$ sor és az $l.$ oszlop elemeit az $e.$ sor $l.$ elemét kivéve. Végül a harmadik lépésben lehet kiszámolni az $e.$ sor $l.$ elemét. Az utolsó két fázis összevonható abban az esetben, ha az első fázis során az $e.$ sor $l.$ elemét egy ideiglenes területre elmentettük. (Ennek a memóriaigénye legfeljebb 8 byte, ami elhanyagolható.)



9. ábra. Az egyes elemek függősége a következő állapot meghatározása során



10. ábra. A következő állapot meghatározását végző implementációk összehasonlítása

A kétféle implementáció összehasonlítása. A két implementációt összehasonlítva (10. ábra) azt tapasztaltam, hogy a kettős pufferelés körülbelül 20-25%-al gyorsabb. Ezt a sebességkülönbséget több dolog együttesen okozza. Egyrészt a memória hozzáférések optimálisabbak, és a szálak közti különbségek (warp divergency) kisebbek abban az esetben, ha egy kernel végzi el az összes számítást. Ezen kívül a kettős pufferelés használata esetén eggyel kevesebb kernelre van szükség, ami azért lényeges, mivel egy kernel elindítása időt vesz igénybe és a plusz egy kernel miatt szükség van plusz egy szinkronizációra is (automatikusan megtörténik), ami szintén időt vesz igénybe. A némileg jobb sebesség ellenére nagy problémák esetén csak az egyszeres pufferelés használható, mivel nagy problémák esetén nincsen annyi memória a videokártyákban, amennyi a kettős pufferelés megvalósításához szükséges lenne. Ez a limitáció egy 1GB memóriával rendelkező videokártya esetén körülbelül akkor jelentkezik, ha $n \cdot m$ legalább 64 millió.

4.3.3. Az algoritmus végének meghatározása

Az iteratív szimplex algoritmus akkor ér véget, ha nem lehet olyan be- és kilépő változót választani, ami kielégítené a feltételeket. (Ha belépő változót nem lehet választani, akkor megtaláltuk az optimális megoldást, ha pedig kilépő változót nem lehet választani, akkor nem létezik véges értékű megoldás.) Ezt a vizsgálatot a be- és kilépő változó pár minden kiválasztása után el kell végezni. A megvalósítására több lehetőség is rendelkezésre áll, viszont méréseim szerint a futásidejük az algoritmus

többi részéhez képest elhanyagolható (körülbelül a teljes futásidő 0.25%-a), ezért nem elemzem részletesen a futásidejüket.

Az első lehetőség, hogy a ki- és belépő változó meghatározása után átmásolom a ki- és belépő változó értékét a GPU memóriájából a CPU memóriájába, majd ez alapján a CPU dönti el, hogy kell-e folytatni az algoritmust (meg kell-e hívni a következő állapot meghatározását végző kernelt).

Ugyanezt az algoritmust úgy is meg lehet valósítani, hogy a be- és kilépő változónak a CPU memóriájába történő átmásolását egy másik végrehajtási sorban (stream) hajtom végre úgy, hogy a másolással egyidőben már elindítom a következő állapot meghatározását végrehajtó kernelt. Ezzel a megoldással gyakorlatilag eltűnik a GPU és a CPU memóriája közötti másolásra fordított idő, viszont le kell kezelni azt esetet, hogy a következő állapot meghatározását végző kernel olyan be- és kilépő változókkal is meg lesz hívva, amelyek azt jelzik, hogy már véget ért az algoritmus (ebben az esetben nem kell semmit csinálnia).

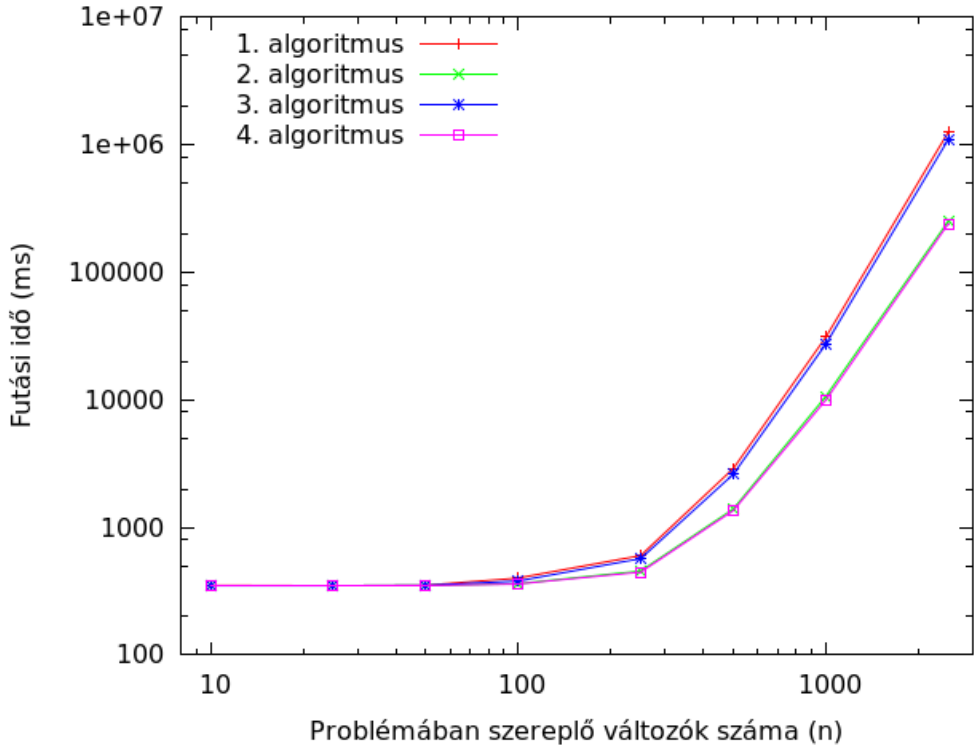
A CUDA 3.5-ös verziót (vagy magasabbat) támogató videokártya használata esetén lehetőség van a dinamikus párhuzamosság használatára is, ami azt jelenti, hogy egy kernel indíthat további kerneleket és szinkronizálhat azok elkészülésére. Ezt kihasználva az iteratív algoritmusnál az algoritmus magját képző iteráció is megvalósítható magán a GPU-n abban az esetben is, ha egy iteráció több kernelből áll. Ezzel a megoldással gyakorlatilag teljesen meg lehet szabadulni a szimplex algoritmus futása közben a CPU és a GPU közötti kommunikációtól. Jelenleg csak az általános célú GPU programozásra fejlesztett videokártyák támogatják a CUDA 3.5-ös verzióját, mint például az Nvidia Tesla sorozat.

Amennyiben a használt videokártya támogatja a dinamikus párhuzamosságot, akkor az azt kihasználó implementációt ajánlott használni, mivel ezzel megszűnik a CPU szerepe az algoritmus futása során, ezért amíg a GPU a lineáris programozási feladatot hajtja végre, addig a CPU egy másik algoritmust hajthat végre. Ha a dinamikus párhuzamosság nem támogatott, akkor pedig a stream-eket használó implementációt célszerű alkalmazni, mivel ha a CPU és a GPU közötti adatkapcsolatot más folyamat is használja, akkor az adatok átmásolása előtt esetlegesen várakoznia kell a rendszernek. Ez elsősorban számítási felhő használata esetén fordulhat elő, mivel ott használja több folyamat is ugyanazt az erőforrást. Nem felhőben lévő számítógép esetén az első két opció közül gyakorlatilag mindegy, hogy melyiket használjuk, mivel nem mérhető ki a két megvalósítás közti futásidő különbség.

4.3.4. A GPU-s algoritmusok futásidejének összehasonlítása

Az eddigiekben bemutattam az algoritmus minden lépésére több lehetséges megvalósítást is. Most ezek közül az alábbi 4 különböző verzió futásidejét fogom összehasonlítani:

1. Egyszeres pufferelés, legnagyobb együtthatójú belépő változó
2. Egyszeres pufferelés, legnagyobb előrelépést biztosító belépő változó
3. Kettős pufferelés, legnagyobb együtthatójú belépő változó
4. Kettős pufferelés, legnagyobb előrelépést biztosító belépő változó



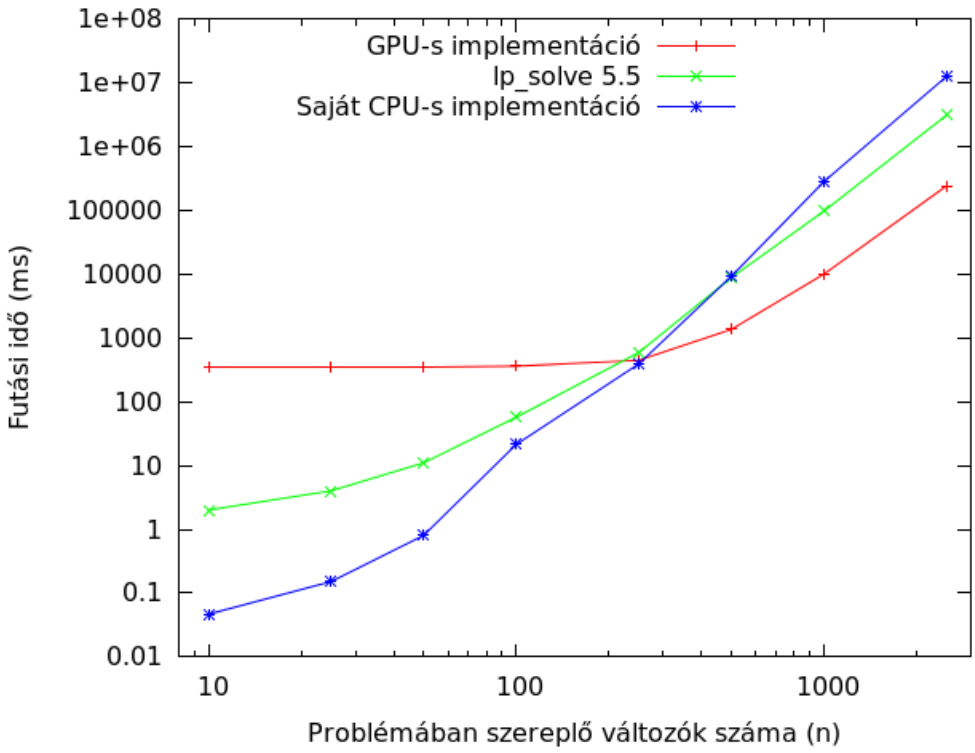
11. ábra. A 4 GPU alapú szimplex implementáció összehasonlítása

Az összehasonlításhoz ugyanazokat a problémákat használom, amelyeket eddig is használtam a futási idők mérésére. Ezekben a problémákban n darab ismeretlen változó és $2 \cdot n$ darab feltétel szerepel és mindegyiknek létezik optimális megoldása. A legnagyobb előrelépést biztosító algoritmus 4 implementációja közül a csempéket használó, de a felesleges szálakat nem eltávolító verziót választottam, mivel az minden bemenő probléma méret esetén jól teljesít. A következő állapot meghatározására pedig mind a két bemutatott algoritmust használom az összehasonlítás során, mivel a kettős pufferelés gyorsabb, viszont bizonyos esetekben csak az egyszeres pufferelés használható.

A négy algoritmus futásideje a 11. ábrán látható. Az ábráról egyértelműen látszik, hogy GPU esetén a legnagyobb előrelépést biztosító belépő változót használó algoritmus futásideje jelentősen kisebb. Szintén megfigyelhető a kétszeres pufferelés nyújtotta sebességjavulás, ez a különbség viszont sokkal kisebb, ezért nem bír akkora jelentőséggel. A grafikon jellegéből az is látszik, hogy a szimplex algoritmus futásideje a vizsgált bemenetekre exponenciálisan skálázódik.

4.4. A CPU-s és a GPU-s algoritmusok összehasonlítása

A GPU alapú algoritmusok hatékonyságát legjobban úgy lehet jellemezni, hogy a futásidejüket összehasonlítom az ugyanazt a problémát megvalósító CPU alapú algoritmusokkal. Az összehasonlításhoz a leghatékonyabb GPU alapú algoritmust (kettős pufferelés, legnagyobb előrelépést biztosító belépő változó választása) hasonlítom össze két CPU alapú algoritmussal. Az egyik CPU alapú algoritmus a saját



12. ábra. A CPU és a GPU alapú szimplex implementációk összehasonlítása

referencia implementációm, amelyik a legnagyobb együtthatójú belépő változót választja, mivel CPU esetén ez a hatékonyabb algoritmus. Ennek az az oka, hogy a legnagyobb előrelépést biztosító belépő változó meghatározása több időt vesz igénybe, mint amennyit a használatával nyerünk. A másik CPU-s implementáció pedig a népszerű nyílt forráskódú lineáris programozás megoldó az lp_solve 5.5 [1]. Ennek a megoldónak nagyon sok különböző paramétert meg lehet adni (például a belépő változó meghatározására használt algoritmust is), de én a futási idők méréséhez az alapértelmezett beállításokat használom.

A futásidő mérésénél minden esetben szöveges formátumban adtam meg az éppen vizsgált algoritmusnak a lineáris programozási probléma leírását (lp_solve 5.5 esetén az alapértelmezett formátumban, saját implementációk esetén pedig egy saját, könnyen feldolgozható szöveges formátumban). Az idő mérésénél az adott program teljes futásidejét mértem a probléma beolvasását és kiírását is beleértve Unix alapú környezetben a beépített time függvénnyel. A probléma beolvasásához és az eredmény kiírásához szükséges idő már kis problémák esetén is gyakorlatilag elhanyagolható a szimplex algoritmus futásidejéhez képest ezért ez nem befolyásolta számottevően a mérési eredményeket.

A különböző algoritmusok futásideje a 12. ábrán látható. A két CPU-s implementációt összehasonlítva jól látszik, hogy a kis problémák esetén a sokkal egyszerűbb saját implementáció teljesített jobban, viszont nagy problémák esetén az lp_solve 5.5 jelentősen gyorsabb. Ennek az az oka, hogy az lp_solve 5.5 tartalmaz egy előfeldolgozót és bonyolultabb algoritmust használ a bázisba belépő változó meghatározására is. Ez a két megoldás nagy problémák esetén jelentősen gyorsítani

tudja a probléma megoldását, viszont kis problémák esetén több időt vesz igénybe, mint amennyit nyerni lehet vele.

A GPU-s implementációt összehasonlítva a CPU-s implementációkkal az látszik, hogy kis problémák esetén nem éri meg a GPU-t használni, mivel az adatok átmásolása és a kernelek elindítása túlságosan sok időt vesz igénybe a teljes futásidőhöz képest. Ennek az időnek a dominanciája okozza azt, hogy kis problémák esetén a GPU-s algoritmus futásideje gyakorlatilag nem változik (mivel a másolás és a kernelek elindítása több idő, mint maga a számítás). A probléma méretének növekedésével viszont a GPU-s implementáció egyértelműen a leggyorsabbá válik. A legnagyobb vizsgált probléma esetén (2500 változó és 5000 feltétel) a saját CPU-s implementáció futásideje körülbelül 50-szer nagyobb, mint a leghatékonyabb GPU-s implementáció futásideje. A GPU-s algoritmus futásidejét az lp_solve 5.5 futásidejéhez hasonlítva azt látjuk, hogy a bizonyos szempontból hatékonyabb algoritmust használó lp_solve 5.5 is több mint egy nagyságrenddel lassabb, mint a GPU-s implementáció.

A saját CPU-s implementáció és az lp_solve 5.5 közötti futásidő különbség nagy problémák esetén körülbelül fél nagyságrend. Ez a különbség azért lényeges, mivel ezt az algoritmusban lévő különbségek okozzák. Ezeknek a különbségeknek egy része implementálható a GPU alapú algoritmusban is, aminek köszönhetően a GPU-s algoritmus sebessége tovább javítható. Várhatóan az így elérhető javulás kisebb, mint a saját CPU-s implementáció és az lp_solve 5.5 közötti futásidő különbség, viszont valószínűleg ezek segítségével a GPU-s algoritmus futásideje akár a felére is csökkenthető.

5. Forgalmi mátrix meghatározása lineáris programozással

Az eddigiekben bemutattam, hogy egy lineáris problémát általánosságban hogyan lehet hatékonyan megoldani GPU segítségével. A következőekben a forgalmi mátrix meghatározásának kérdésével fogok foglalkozni olyan megközelítésből, hogy hogyan lehet az eddig bemutatott technikákat erre a konkrét műszaki problémára alkalmazni.

5.1. A lineáris programozás probléma meghatározása

A 3.2. fejezetben bemutattam, hogy hogyan lehet modellezni a forgalmi mátrix meghatározásának a problémáját. Most azt a modellt felhasználva bemutatom, hogy hogyan lehet ugyanezt a problémát lineáris programozási problémaként meghatározni.

A linkterhelésekből származó mátrix egyenlet ($R \cdot X = L$) egy lineáris egyenletrendszert reprezentál, amit mindenféle módosítás nélkül bele lehet tenni a lineáris programozási probléma feltételei közé. Ahhoz, hogy megfeleljen a szimplex algoritmus sztenderd alakjának, az egyenletrendszert fel kell bontani a következő két részre:

$$R \cdot X \geq L$$

$$-1 \cdot R \cdot X \geq -1 \cdot L$$

A két vektor közötti egyenlőtlenséget elemenkénti egyenlőtlenségként kell értelmezni.

A linkterhelésekből származó egyenleten kívül az f függvény minimalizálását, mint optimalizálási célt kell beleépíteni a lineáris programozási problémába. A legegyszerűbb eset az, ha az f függvény a számított forgalmi mátrix (X) lineáris függvénye, mivel ebben az esetben egyszerűen meg lehet adni az f függvényt az optimalizálási feltételnek. Ez az eset viszont semmilyen jól használható f esetén sem áll fent ezért a következőkben bemutatok néhány különböző f függvényt és mindegyikre egy lehetséges megvalósítást.

Az egyik lehetőség, hogy definiálok egy hibafüggvényt, amit a forgalmi mátrix összes elemére alkalmazok és az optimalizálás során a legnagyobb hibát (C) próbálom meg minimalizálni. Ilyen függvényeknek egy csoportja leírható az alábbi formában, ahol $A_{i,j}$ a becsült, $X_{i,j}$ pedig a számított forgalmi mátrix egy elemét jelöli, q pedig egy konstans:

$$\frac{|A_{i,j} - X_{i,j}|}{(A_{i,j})^q} \leq C$$

Ez a függvény az abszolút érték képzés miatt nem lineáris, viszont könnyen felbontható az alábbi két lineáris feltételre, amelyeknek egyszerre kell teljesülnie:

$$\frac{A_{i,j} - X_{i,j}}{(A_{i,j})^q} \leq C$$

$$\frac{A_{i,j} - X_{i,j}}{(A_{i,j})^q} \geq -C$$

A q paraméter értékének különböző megválasztásával ennek a függvénynek különböző jelentéseket lehet adni. Ha $q = 0$, akkor az abszolút hibának felel meg, ha pedig $q = 1$, akkor a becsült értékhez viszonyított relatív hibának. A q értékének ezen két szélső érték közötti változtatásával be lehet állítani, hogy mi az a metrika, amelyik szerint a számított mátrixot a becsült mátrixhoz közelíteni szeretnénk. Kis q érték választása esetén a kis értékek relatív hibája a többi értékhez képest viszonylag nagy lesz. Nagy q érték esetén pedig a nagy értékek abszolút hibája lesz relatív nagy a többi értékhez képest.

A legnagyobb hiba minimalizálása gyakran nem vezet túl jó eredményre, mivel néhány hibás adat nagyon könnyen meg tudja növelni a legnagyobb hiba értékét. Ez elkerülhető azzal, ha a legnagyobb hiba helyett a hibák abszolút értékének összegét próbáljuk meg minimalizálni. Ez megoldható úgy, ha az előző esethez képest a következő célfüggvényt és feltételeket használjuk (az abszolút érték az előbbiekben bemutatott átalakítással eltüntethető):

$$\min \sum_{i,j} C_{i,j}$$

$$\frac{|A_{i,j} - X_{i,j}|}{(A_{i,j})^q} \leq C_{i,j}$$

Az optimalizálandó függvény megválasztásával ezeken az eseteken kívül is nagyon sokféleképpen lehet paraméterezni az optimalizálási folyamatot. Például van arra is lehetőség, hogy a két oldali hibát különböző mértékben vegyük figyelembe, vagy akár arra is, hogy a hibát valamilyen nem lineáris függvény törött vonalas közelítésével számoljuk. Ezeket a feltételeket valósítja meg a következő feltételrendszer:

$$\frac{A_{i,j} - X_{i,j}}{(A_{i,j})^q} \leq C_{i,j}$$

$$\frac{A_{i,j} - X_{i,j}}{(A_{i,j})^q} \geq -\frac{1}{\alpha} \cdot C_{i,j}$$

$$\frac{A_{i,j} - X_{i,j}}{(A_{i,j})^q} \leq \frac{C_{i,j}}{\beta} + \gamma$$

$$\frac{A_{i,j} - X_{i,j}}{(A_{i,j})^q} \geq -\frac{1}{\alpha} \cdot \left(\frac{C_{i,j}}{\beta} + \gamma \right)$$

Ebben a feltételrendszerben amennyiben a számított érték kisebb, mint a becsült érték, akkor azt a hibát α-szor akkora súllyal veszem figyelembe. Ha pedig a hiba abszolút értéke nagyobb mint γ, akkor a hiba γ-nál nagyobb részét β-szor akkora súllyal.

A fentiek alapján jól látszik, hogy összetett feltételeket is meg lehet fogalmazni az optimalizálás során. Ezt kihasználva, ha rendelkezésünkre áll az az információ, hogy a kiszámított forgalmi mátrixot mire fogják felhasználni, akkor van arra lehetőség, hogy a felhasználás szempontjából kritikus értékeket határozzuk meg úgy, hogy az eredeti becsléshez közeli értéket kapjunk. Az optimalizálási feltétel bonyolultságának növelésével viszont a szükséges számítási kapacitás is megnő, annak ellenére, hogy nem kellett újabb ismeretleneket bevezetni a feltételek megfogalmazásához. Ennek az az oka, hogy a törött vonalas közelítés miatt a megoldandó problémában szereplő feltételek száma közel a duplájára nőtt.

5.2. A lineáris programozási probléma értékelése

Az előbbiekben vázolt modell kipróbálásához a valódi IP hálózatok szerkezetéhez nagyon hasonló szerkezetű hálózatokat generáltam. Feltettem, hogy a hálózatban az egyes csomópontok közötti kommunikáció mindig a legrövidebb út mentén történik. Ezt követően a hálózathoz tartozó forgalmi mátrixot úgy generáltam le, hogy a forgalmi adatok eloszlása megfeleljen a hálózati tomográfián alapuló megoldásoknál használt eloszlásoknak. A forgalmi mátrix és a hálózatban szereplő utak ismeretében egyszerűen (és determinisztikusan) ki lehet számolni a hálózat egyes éleinek terhelését. A hálózat paramétereinek meghatározása után a forgalmi mátrix becslését úgy állítom elő, hogy a pontos forgalmi mátrix minden egyes értékét megszoroztam egy normál eloszlású (1 középértékű, $(0; 2)$ intervallumra korlátozott) zajjal. A becsült forgalmi mátrix, a linkterhelések és a hálózati topológia alapján kiszámított forgalmi mátrixot végül az eredeti kiindulási értékeket tartalmazó forgalmi mátrixhoz hasonlítottam.

A bemenő adatok ilyen módon történő generálásának az előnye, hogy a becsült és a számított forgalmi mátrix hibája is kiszámítható és ezzel a használt algoritmus értékelhető. Az általam használt algoritmus a méréseim szerint a forgalmi mátrix becslés hibáját kis mértékben (körülbelül 5%-al) rontotta, viszont az így kiszámított mátrix már eleget tesz a linkterhelésekből származó feltételeknek, ami az eredeti becsült mátrixról szinte soha sem mondható el. A linkterhelésekből adódó egyenletek kielégítése azért fontos, mivel ezzel biztosítható az, hogy a különböző adatok koherensek legyenek, azaz ne mondjanak egymásnak ellen. A koherencia az adatok további feldolgozása során bír komoly jelentőséggel, mivel az egymásnak ellentmondó adatok sokat ronthatnak a különböző numerikus számítások stabilitásán.

6. Összefoglalás

A dolgozatomban bemutattam, hogy hogyan lehet a szimplex algoritmust hatékonyan implementálni GPU-ra a CUDA segítségével. A szimplex algoritmus fontosabb lépéseinél megvizsgáltam több lehetséges algoritmikust és implementációs megoldást is az optimális kiválasztásához. Ennek köszönhetően sikerült egy olyan implementációt összeállítanom, amelyik nagy problémák esetén a saját CPU alapú referencia implementációmnál körülbelül 50-szer gyorsabb. A népszerű lp_solve 5.5 nevű programnál pedig körülbelül 12-szer gyorsabb annak ellenére, hogy sokkal egyszerűbb algoritmust és kevésbé optimalizált kódot használ. Ezek a sebességkülönbségek elég nagyok ahhoz, hogy legyen értelme a gyakorlatban jelentkező lineáris programozási feladatokat a GPU segítségével megoldani, mivel az elérhető sebességnövekedés legtöbb esetben átváltható a pontosság növelésére. A GPU-s implementációm skálázódását leginkább korlátozó tényező a GPU-ban rendelkezésre álló memória mennyisége (általában 1GB), mivel a hatékony működéshez arra van szükség, hogy a teljes probléma beleférjen a GPU memóriájába.

A lineáris programozás használhatóságát egy hálózattervezésből származó példán, a forgalmi mátrix meghatározásán illusztráltam. A jelenleg használt megoldásokhoz képest, arra mutattam be egy lehetséges megoldást, hogy ha már rendelkezünk egy olyan becsléssel a hálózat forgalmi mátrixát illetően, ami nem koherens a hálózat egyes linkjein átáramló forgalom mennyiségével, akkor hogyan tudunk meghatározni egy olyan forgalmi mátrixot, amelyik nagyon hasonlít az eredeti becslésünkre és még a linkterhelésekből adódó egyenleteket is kielégíti.

A szimplex algoritmus GPU-s implementációjában a dolgozatomban bemutatott részeken kívül még nagyon sok lehetőség rejlik. Egyrészt a bázisba be- és kilépő változók kiválasztására léteznek a bemutatott algoritmusoknál hatékonyabb, de sokkal bonyolultabb algoritmusok is amiket érdemes lehet GPU-ra is implementálni. Ezen kívül a valós problémák esetén egy gyakori eset az, hogy a feltételeket tartalmazó mátrix ritka (kevés nem 0 elemet tartalmaz). A mátrixnak ezt a tulajdonságát ki lehet használni az algoritmus memória igényének csökkentésére és sebességének növelésre. Mivel a szimplex algoritmust a gyakorlatban nagyon sokat használják, ezért az itt felsoroltakon kívül még számos kisebb nagyobb trükk létezik amikkel az algoritmus futásideje csökkenthető. Ezek egy része biztos, hogy nem implementálható hatékonyan GPU segítségével, viszont a többi optimalizáció GPU-s implementációja további sebességnövekedést eredményezhet.

7. Rövidítésjegyzék

BLAS	Basic Linear Algebra Subprograms
CBLAS	C Basic Linear Algebra Subprograms
CPU	Central Processor Unit
CUBLAS	CUDA Basic Linear Algebra Subprograms
CUDA	Compute Unified Device Architecture
ECMP	Equal Cost Multi-Path
GNU GSL	GNU Scientific Library
GPGPU	General Purpose Graphical Processor Unit
GPU	Graphical Processor Unit
PFLOPS	Peta floating point operation per second
TFLOPS	Tera floating point operation per second

8. Hivatkozások

[1] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. *lpsolve : Open source (Mixed-Integer) Linear Programming system.*

[2] Jin Cao, Drew Davis, Scott Vander Wiel, Bin Yu, Scott Vander, and Wiel Bin Yu. Time-varying network tomography: Router link data. *Journal of the American Statistical Association*, 95:1063–1075, 2000.

[3] Pedro Casas and Sandrine Vaton. On the use of random neural networks for traffic matrix estimation in large-scale ip networks. In *Proceedings of the 6th International Wireless Communications and Mobile Computing Conference*, IWCMC '10, pages 326–330, New York, NY, USA, 2010. ACM.

[4] Mark Galassi, Jim Davies, James Theiler, Brian Gough, Gerard Jungman, Michael Booth, and Fabrice Rossi. *Gnu Scientific Library: Reference Manual.* Network Theory Ltd., February 2003.

[5] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010. Version 1.7.0.

[6] Dingde Jiang, Xingwei Wang, Lei Guo, Haizhuan Ni, and Zhenhua Chen. Accurate estimation of large-scale {IP} traffic matrix. *{AEU} - International Journal of Electronics and Communications*, 65(1):75 – 86, 2011.

[7] Dingde Jiang, Zhengzheng Xu, Hongwei Xu, Yang Han, Zhenhua Chen, and Zhen Yuan. An approximation method of origin–destination flow traffic from link load counts. *Computers and Electrical Engineering*, 37(6):1106 – 1121, 2011.

[8] Jacek P. Kowalski and Bob Warfield. Modelling traffic demand between nodes in a telecommunications network. In *ATNAC'95*, 1995.

[9] Derek Lam, Donald C. Cox, and Jennifer Widom. Teletraffic modeling for personal communications services. *IEEE COMMUNICATIONS MAGAZINE*, 35:79–87, 1997.

[10] A. Medina, N. Taft, K. Salamatian, S. Bhattacharyya, and C. Diot. Traffic matrix estimation: existing techniques and new directions. *SIGCOMM Comput. Commun. Rev.*, 32(4):161–174, August 2002.

[11] Nvidia. *CUBLAS Library User Guide.* nVidia, v5.0 edition, October 2012.

[12] Nvidia. Parallel programming and computing platform. `http://www.nvidia.com/object/cuda_home_new.html`, 2013. 10. 10.

[13] Matthew Roughan, Albert Greenberg, Charles Kalmanek, Michael Rumsewicz, Jennifer Yates, and Yin Zhang. Experience in measuring backbone traffic variability: models, metrics, measurements and meaning. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, IMW '02, pages 91–92, New York, NY, USA, 2002. ACM.

[14] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

[15] Y. Vardi. Network Tomography: Estimating Source-Destination Traffic Intensities from Link Data. *Journal of the American Statistical Association*, 91(433):365–377, March 1996.

[16] Yin Zhang, Matthew Roughan, Nick Duffield, and Albert Greenberg. Fast accurate computation of large-scale ip traffic matrices from link loads. *SIGMETRICS Perform. Eval. Rev.*, 31(1):206–217, June 2003.