Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Médiainformatikai Tanszék

Szőts Ákos

Nagy hálózatokban zajló jelterjedési folyamatok szimulációjának felgyorsítása GPU segítségével

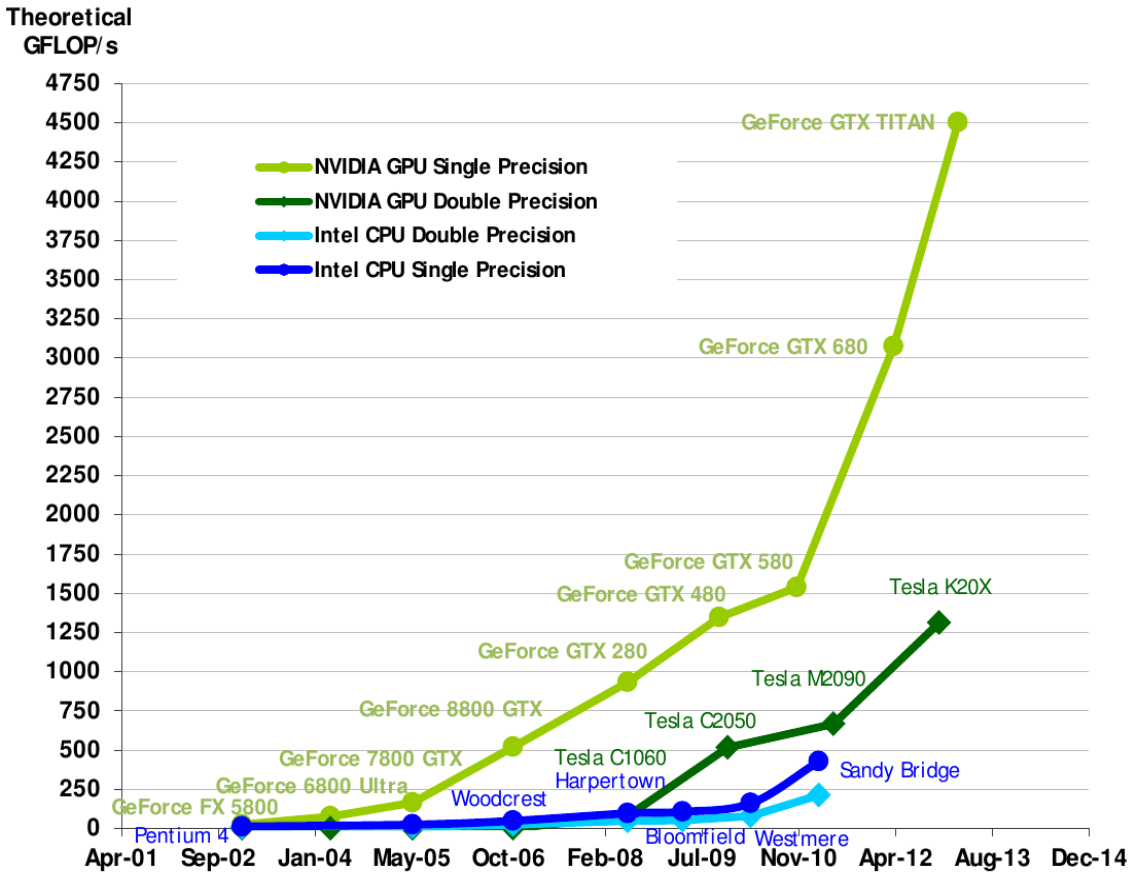
Konzulensek
Dr. Gulyás András
Szalay Kristóf

BUDAPEST, 2013.

Tartalomjegyzék

Bevezető

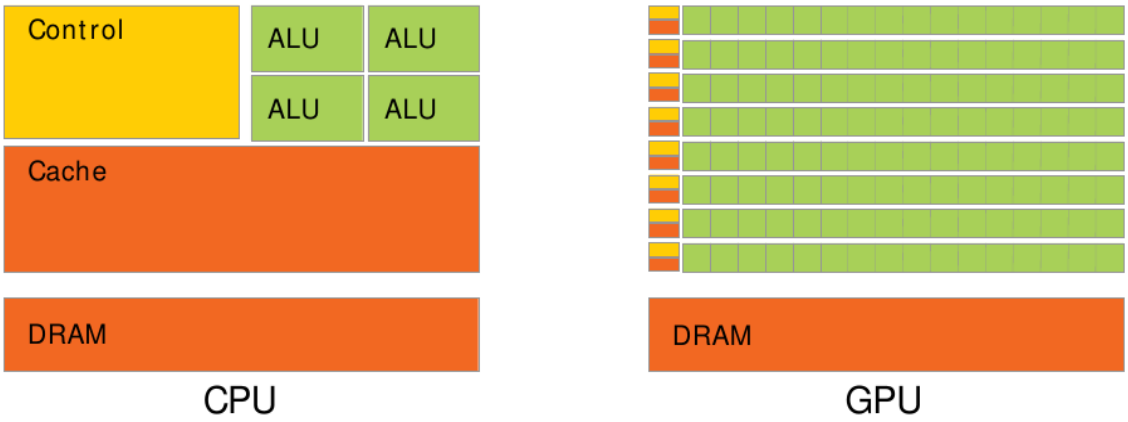
Egyre több és több olyan kutatás lát napvilágot, ami korábban elképzelhetetlen mennyiségű adattal dolgozik, és a bennük foglalt számítási problémák is egyre nagyobbá és összetettebbé válnak. Ezzel szemben a hagyományos CPU technológia jelenleg nem skálázódik olyan mértékben, ami ezeket a megnövekedett igényeket kielégítené [1].



1. ábra: Lebegőpontos műveletek másodpercenként CPU-n és GPU-n

A grafikus feldolgozóegység (*Graphics Processing Unit – GPU*) lehetővé teszi a több ezer kisebb részfeladatra felbontott komplex feladatok párhuzamos végrehajtását, amivel több nagyságrenddel gyorsíthatóak a számítási feladatok. Az 1. ábrán látható, ahogyan az idő előrehaladtával és a technológia fejlődésével folyamatosan nő a videokártyák számítási teljesítménye, mind az egyszeres-, mind pedig a dupla pontosságú lebegőpontos műveletek esetében.

A CPU és a GPU közötti ezen eltérés oka, hogy a videokártyák számításigényes, nagyon jól párhuzamosítható feladatokra lettek tervezve – pontosan amiről a grafikus renderelés szól –, így a legtöbb tranzisztort az adatfeldolgozásra szánták, nem pedig az adatok gyorsítótárazására és a vezérlésirányításra [2], ahogyan ezt a 2. ábra is mutatja:



2. ábra: A CPU és a GPU alapvető felépítése: a GPU több tranzisztort szentel az adatfeldolgozásra

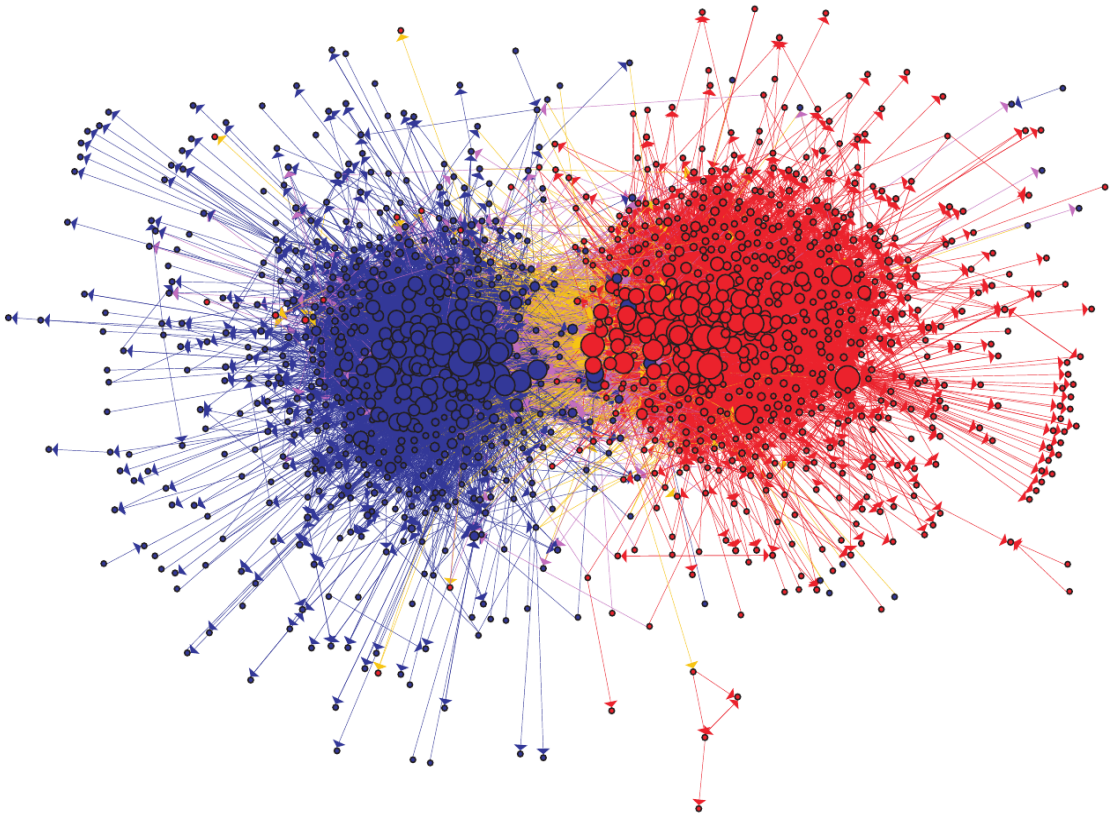
Az NVIDIA 2006-ban létrehozta a CUDA (*Compute Unified Device Architecture*) párhuzamos számítási platformot, amely segítségével C, C++ vagy Fortran nyelven vezérelhetővé vált a videokártya; így többé már nem volt szükség OpenGL vagy DirectX árnyalók (*shaderek*) írására a GPU-k általános célú programozásához (*General-purpose computing on graphics processing units – GPGPU*) [3].

Az a tény, hogy a megfelelően párhuzamosítható problémák megoldása nagyságrendekkel gyorsult, és ehhez a videokártyák felépítésén túl mindössze a C nyelv néhány kiegészítését kellett megismerni, számos tudományterület érdeklődését felkeltette. Sok tudományos CUDA alkalmazás készült már bioinformatika, kémia, pénzügy, folyadékdinamika, szerkezeti mechanika, katonai védelem, orvosi képfeldolgozás, illetve időjárás- és klímakutatás témakörében [4].

Ehhez az impozáns felsoroláshoz szeretne csatlakozni a Turbine a dinamikushálózat-analízis területével.

1 Dinamikushálózat-analízis

A terrorista szervezetek olyan speciális hálózati felépítéssel rendelkeznek, ami a legtöbb hierarchikus szervezettől eltér: sejtszerűek és elosztottak. Bár a legtöbb parancsnoknak vagy ügynöknek legalább intuitív megérzése van ezen hierarchiák működéséről, és hogy hogyan tudják befolyásolni azokat, de már sokkal kevésbé tudják megmagyarázni, hogy az ilyen dinamikusan változó hálózatok hogyan fejlődnek, változnak, alkalmazkodnak és hogyan lehet őket destabilizálni. Legegyszerűbb megoldásnak az tűnhet, ha egyszerűen megtalálják a kapcsolatokat, amik elvezetnek a hálózat centrális, központi figurájához. De hogyan lehet ezeket a kapcsolatokat feltérképezni sok pontatlan, elavult vagy félrevezető információ alapján? Még az sem biztos, hogy a kulcsszereplő eltávolításával a hálózatot sikerül felbontani; az is elképzelhető, hogy ugyanaz lesz a hatása mint a hidra levágott fejének, azaz több új vezető lép a volt helyébe. A hálózat *változásainak hatását* hiba nem vizsgálni ilyen esetekben [5].



1.1. ábra: Amerikai politikai blogok közösségi felépítése és kapcsolatai 2005-ben. Kékkel a liberálisok, pirossal a konzervatívok jelölve. A narancssárga kapcsolatok a liberálisból a konzervatívba, a lilák pedig vice-versa irányulnak. Minél nagyobb egy blog, annál többen hivatkoznak rá.

Ahogyan a legutóbbi két amerikai elnökválasztás megmutatta, az internet használata valódi kulcs lehet a győzelemhez. Hogyan lehet ezt a folyamatosan változó információhalmazt feltérképezni és szimulálni a következő lépésüket? Az 1.1. ábrán

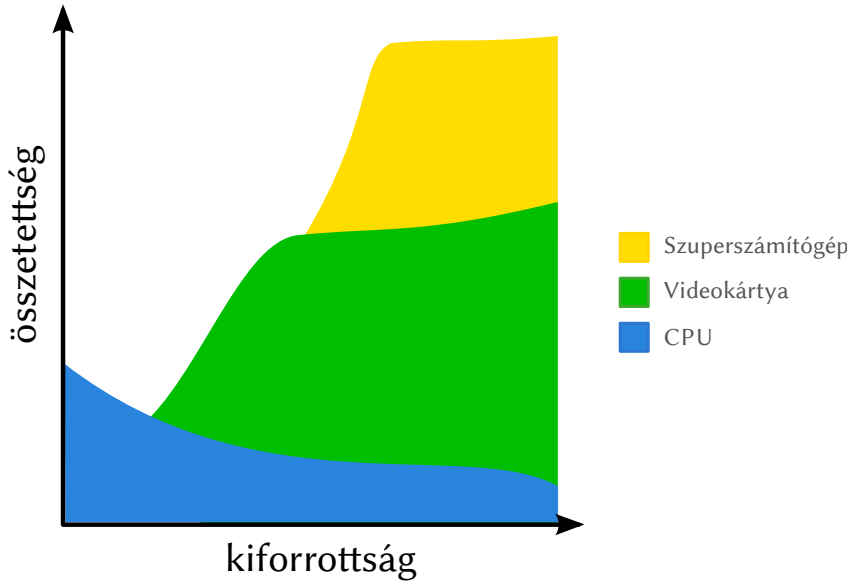
látható az amerikai politikai blogoszféra a köztük lévő kapcsolatokkal 2005 elején [6]. Miként lehet a köztük kialakuló kapcsolatok változásait előrejelezni? Hogy épülnek ki ezek a közösségek és milyen társalgások zajlanak közöttük? Lehet-e befolyásolni ezen hálózatokat?

A dinamikushálózat-analízis (*dynamic network analysis – DNA*) a hálózatok változásaival (azaz dinamikájával) foglalkozik. A fentebb említetteken kívül rengeteg más szakma is hasznát veszni ennek az interdiszciplináris tudományterületnek. A *közgazdaságtanban* a nemzetközi kereskedelem, a portfólióoptimalizálás, ipari szerveződések, a *szociológiában* a Facebookhoz hasonló szociális hálók vagy a szervezetelmélet (organizational theory), a *biológiában* a neurális- vagy protein hálók, a *számítástudományban* az útkeresési algoritmusok és végül a *mérnöki területek* között az energiaellátás, a számítógépes hálózatok, a telekommunikáció és a közlekedés tudományterülete mind-mind hálózatokra épül [7]. Ezen hálózatok nagyban hasonlítanak egymásra abból a szempontból, hogy mindegyikük lehet nagyon kiterjedt és topológiájukban összetett, illetve néhányuk egymásra is kihathat (pl. a közlekedés és a telekommunikáció).

Barabási Albert-László és Albert Réka 1999-ben néhány kollégájával együtt nekiláttak a világhálón lévő weboldalak közti kapcsolatok felderítéséhez. Amikor a keresőrobotjaik eredményeiből kirajzolták a kapcsolati hálót, meglepődve tapasztalták, hogy a dokumentumok nem egyformán népszerűek, hanem volt néhány olyan, amikre sokkal több link vezetett [8]. Ez a felfedezés alapozta meg a skálafüggetlen hálózatok tanulmányozását; sikerült bebizonyítaniuk, hogy az organikusan felépült hálózatok hasonlítanak egymásra [9].

Egy hatalmas szociális háló több millió csomópontból állhat, míg köztük több száz vagy ezer millió kapcsolat is elképzelhető. Ekkora méretű problématér szimulációja órákig vagy napokig tarthat, amire már egy egyszerű számítógép nem hatékony vagy akár nem is elegendő. Szerencsére ezen, skálafüggetlen hálózatok sajátja, hogy a megfelelő paramétereket megtartva egy hozzájuk nagyon hasonló, de jóval kisebb variáns is elkészíthető belőlük [9], ami az otthoni gépen már szimulálható.

Egy olyan szuperszámítógépet (mint a BME Superman) ami már tudná a nagy hálózatot is szimulálni, sokan szeretnék igénybe venni és így gyakran várakozási sor alakul ki a rajta futtatandó programok között, illetve a kész programot nagyon nehéz helyben módosítani, ezért csak letesztelt, kiforrott kódot érdemes rajta futtatni. Ennek okáért szükséges egy olyan környezetet találni, ami az ilyen szuperszámítógépekben is megtalálható, de olcsóbb és így otthon is tesztelhetővé válik a gyorsított program. Ez a CUDA. Már a legolcsóbb, pár ezer forintos GeForce videokártyák is képesek megteremteni ezt a környezetet, így az otthoni fejlesztés és tesztelés is kivitelezhetővé válik.



1.2. ábra: Javasolt eszköz a szimuláció futtatására a program kiforrottságát és a hálózat összetettségét figyelembe véve

Ahogyan az 1.2. ábrán látható, egy hálózatszimulátor fejlesztésének kezdetén egyszerűbb CPU-n tesztelni az alakuló programot alig összetett hálózatokon, majd pedig minél teljesebbé és teszteltebbé válik a szimulátor, érdemesebb átültetni videokártyára, hogy a nagyobb méretű hálózatokat is tudja kezelni. Majd mikor a GPU-ra való átültetés is sikeres volt, ezt a programot át kell mozgatni egy szuperszámítógépben lévő nagy teljesítményű GPU-ra, hogy az igazán összetett felépítésű hálózatokon is gyorsan és hatékonyan tudjon dolgozni.

2 Turbine

A Turbine egy olyan, Creative Commons licenc alatt terjesztett nyílt hálózatanalízis csomag, amely a hálózatok különféle változásait hivatott vizsgálni [10].

E C++ nyelven íródott program többek között képes különféle hálózatokat létrehozni, azokon megadott modellek alapján szimulációt futtatni, miközben modellezi a hálózaton kívüli változások (perturbációk) hatásait.

2.1 Alapvető felépítés

A Turbine egy több összetevőből álló programcsomag. Tartalmaz eszközöket különböző felépítésű hálózatok létrehozására, megtekintésére vagy azok más alkalmazásokból történő konvertálására, kezdőállapotok és perturbációk különféle függvények szerinti elkészítésére és az eredményhalmaz vizsgálatára is.

Támogatja a különféle pluginokkal történő kiegészítését, amik megosztott könyvtárként kerülhetnek megvalósításra. Ezekben az .so/.dll fájlokban meghatározott függvényszignatúráknak kell elhelyezkedniük, és ekkor a fő program a könyvtár betöltése után meg tudja hívni őket, így kiegészítve saját tudását. Többek között ilyen felépítéssel vannak megvalósítva a szimulációk (minden modell egy-egy modul), a hálózatkonvertálás (minden ki- és bemenő formátum egy megosztott könyvtár), illetve a hálózatgenerálás és -normálás is.

A program indítása és a megfelelő adatfájlok betöltése után a Turbine a perturbációkat, a kiinduló vektorokat, a hálózatot és az ezeket körülfogó szimulációs tömböt építi fel.

A szimulációs modellt tartalmazó struktúrában foglalnak helyet a csúcsok és élek vektorként ábrázolva, egyedi, növekvő azonosítóval, illetve a hozzájuk szorosan kapcsolódó állapotok is.

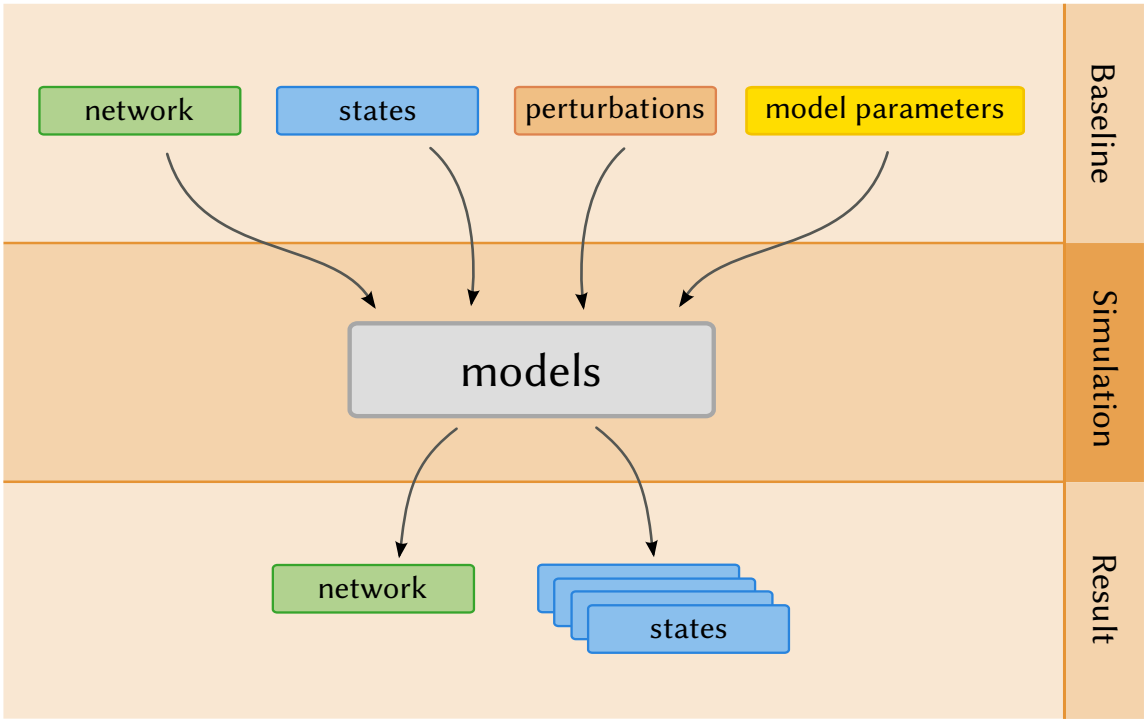
A kiindulási állapotok és a perturbációk egy kétdimenziós tömbben tárolódnak, ahol a függőleges tengely az idősíkot jelképezi, míg a vízszintes tengely a node-ok/linkek számát; azaz a szimuláció minden egyes lépéséhez hozzá van társítva a megfelelő számú csúcs/él. Mivel a Turbine néhány modellje lehetőséget ad arra, hogy a csomópontok száma és a köztük lévő kapcsolatok megváltozzanak, ezért az ezeket tároló x tengelyt leképező tömb hossza változó lehet (*ragged array*).

Ezen struktúrák feltöltése után elindul a szimuláció, ami a paraméterként megadott modell szerint folyamatosan módosítja a csomópontokhoz és élekhez rendelt állapotokat, akár egymás után több százszor vagy több ezerszer. Ebből pedig látható, hogy ez az a jól párhuzamosítható pont, ahol érdemes elkezdeni a videokártyára való átültetést. Ehhez pedig, ahogyan a következő fejezetekben látható lesz, meg kell vizsgálnunk egy GPU általános felépítését, ki kell számolnunk, hogy egyáltalán érdemes-e elkezdeni az átültetést, ha igen, akkor mire kell figyelni eközben, végül pedig meg kell mérni, hogy mekkora teljesítménynövekedést sikerült elérnünk.

A szimuláció lefolyásáról a következő fejezet ad bővebb információt, míg a korábban említett struktúrák részletes leírásai a CUDA nyelvre történő átalakításukkor, a 6. fejezetben kerülnek kifejtésre.

2.2 A szimuláció elemei

Ahhoz, hogy a CUDA nyelvre való átalakítást és az erre épülő későbbi méréseket minden részletre kiterjedően el tudjuk végezni, szükséges megismerni a Turbine szimulációjának alapvető működését.



2.1. ábra: A Turbine szimuláció felépítése

Egy szimuláció időbeli lefolyása három fő részre bontható: a kezdeti állapotban a program segítségével el kell készíteni a hálózatot vagy gráfot (*network*), amin a szimuláció folyni fog. Minden gráf csúcsokból (*node*) és a köztük lévő élekből (*link*) áll. A hálózatokról bővebben lásd a 2.3. fejezetet. Ezen a hálózaton lehet a szimulációt futtatni, majd a kapott a végeredményt elmenteni. A most felsorolandó elemek mindegyike egy-egy fájlként jelenik meg a programban. Ezek létrehozása és beolvasása a Turbine feladatai közé tartozik, azonban részletes leírásuk kívül esik ezen dolgozat témáján.

Az ábrán látható, a hálózat utáni következő elem a csomópontokhoz és az élekhez egyaránt társítható állapotváltozók (*States*). Ilyen lehet például egy csomópont energiája, ami minden egyes ciklusban meghatározott mennyiséggel csökken. Ezen változók bemenetként a kezdeti állapot megadására szolgálnak, ebből indul ki a szimuláció első lépése. A szimuláció közben, minden egyes ciklus végén ezek elraktározódnak a célból, hogy később a változások visszajátszhatók legyenek.

A perturbációk (*perturbations*) vagy külső változások olyan kezdeti gerjesztések, amik a modelltől függetlenül jelen lehetnek és befolyásolják a végeredményt, így imitálnak külső hatásokat. Alkalmazhatók mind a node-okra, mind pedig a linkekre, akár szelektíven néhányra is. Továbbá beállítható az is, hogy hány lépésen (cikluson) keresztül érvényesüljön a hatásuk.

Elérkeztünk a legfontosabb részhez, a szimuláció magjához, a modellekhez. A modellekben összpontosul minden eddig leírt bemenő paraméter, ami alapján az a megadott lépésszámszor végzi el a szimulációt. Legtöbbjük működését kívülről is lehet konfigurálni egyéb bemenő értékekkel (*model parameters*).

A Turbine több modellt támogat, amik mind mást szimulálnak az adott hálózaton. Létezik a gráfon hullámterjedést szimuláló változat; olyan, ami információterjedést vizsgál elhalási faktorral (pl. pletykaterjedés) vagy a közlekedőedények felépítését és bennük a víz áramlását imitáló modell (az ún. *vessels* modell) is. Bővebb leírásukat l. a 2.4. fejezetben.

A szimuláció lefutásának legvégén a Turbine az esetleg megváltozott hálózatot lemezre írja a minden lépésben rögzített állapotok pillanatképeivel együtt.

A fentiekből észrevehető, hogy a folyamatból legtöbb időt a szimuláció több száz- vagy több ezerszeri futtatása teszi ki, így elsősorban itt érdemes elkezdeni az optimalizációt.

2.3 Hálózattípusok

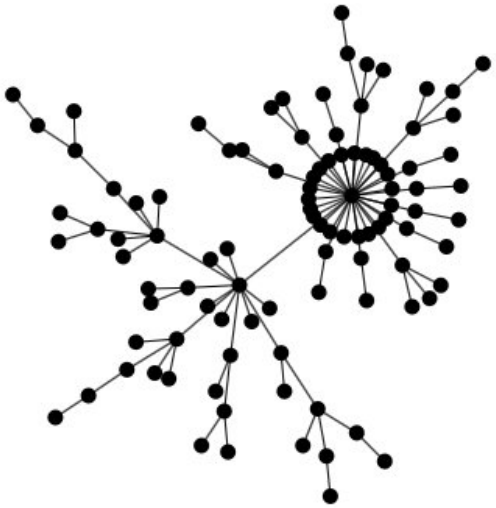
A program több hálózattípust támogat, amik felépítése más és más. Mivel a videokártyára való átültetés után megvizsgáljuk, hogy mekkora gyorsulást sikerült elérni, fontos tisztában lennünk a különféle gráfok alapvető felépítésével, hogy meg tudjuk mérni, hogy a szimulációs sebesség függ-e tőlük.

2.3.1 Barabási–Albert-gráf

A Barabási–Albert-modell egy olyan gráffejlődési algoritmust ír le, ahol egy, kezdetben legalább két csomópontú gráfhoz folyamatosan új csúcsokat adunk, és ezeket egy-egy éllel kapcsoljuk hozzá tetszőlegesen kiválasztott számú, véletlenszerű régi csúcshoz. Ezen kiválasztás során prioritást élveznek azok a régi csúcsok, amiknek fokszáma nagyobb a többinél [11].

A gráf véletlen gráf, kisvilág-tulajdonsággal.

A Turbine-ban használt elnevezése: *grown*.



2.2. ábra: Barabási–Albert-gráf

2.3.2 Watts–Strogatz-gráf

2.3. ábra: Watts–Strogatz-gráf

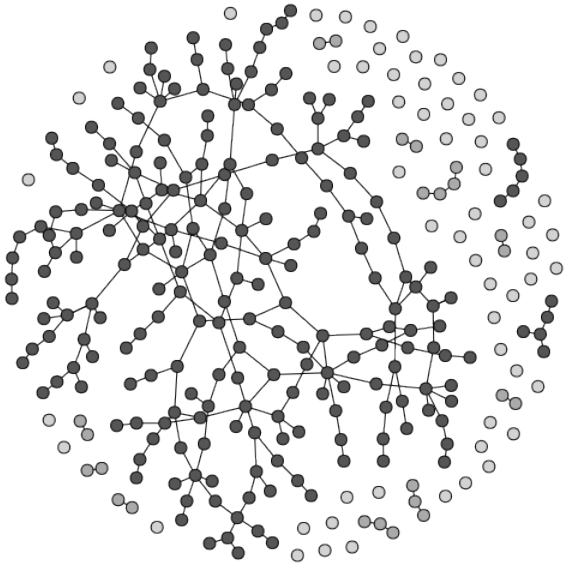
Ez a szintén véletlen, kisvilág-tulajdonsággal rendelkező gráf, amelyben kezdetben N csomópont található egy gyűrű alakú rácsban, amik közül mind K szomszédjukhoz vannak kötve, majd egyenletes eloszlással egy β valószínűséggel az algoritmus „újraköt” minden csúcsot [12].

A Turbine-ban használt elnevezése: *smallworld*.

2.3.3 Erdős–Rényi-gráf

Ez a modell egy véletlen gráfot hoz létre, ahol a benne lévő N csúcs egymástól függetlenül, p valószínűséggel kapcsolódik egymáshoz [13].

A Turbine-ban használt elnevezése: *random*.



2.4. ábra: Erdős–Rényi-gráf

2.3.4 Rácsháló

Ez a legegyszerűbb elérhető gráffajta a programban. Egy n oszlopú, m sorú hálót hoz létre, benne n×m csomóponttal, és ha létezik, akkor a felső, jobb, alsó és bal oldali szomszédokkal való összekötöttséggel.

A Turbine-ban használt elnevezése: *lattice*.

2.4 Modellek

2.4.1 Communicating vessels

A Turbine több modellt támogat, amik szerint a hálózat szimulálható. Ebben a dolgozatban a *communicating vessels* (közlekedőedények) modellje kerül átalakításra CUDA nyelvre, így szükséges ennek részletesebb megismerése.

A vessels modell egy állapotváltozóval, az energiával dolgozik, ami minden csomópont sajátja. Itt a csomópontok az edényeket, míg a köztük lévő kapcsolatok a hajszálcsöveket jelképezik. Ezen modell mögötti alapvető elképzelés az, hogy az intenzív (azaz a rendszer nagyságától független) fizikai mennyiségek, mint a hőmérséklet, nyomás, sűrűség, a közlekedőedényekben áramló folyadékhoz hasonló kiegyenlítődésre törekszenek. Használható továbbá az ellenállás hálózatokban a feszültségek alakulásának modellezésére vagy a véletlen bolyongások végződésének várható értékének számításához [10].

Az algoritmus a következőképpen működik: minden szimulációs lépésben minden csúcs a benne lévő energia egy részét átadja a kapcsolatain keresztül a többi csomópontnak. Ennek a mértéke arányos az időléptékkel, a csövek átmérőjével (az él súlyaként jelenik meg) és az él nyomásával (a link két oldalán lévő node energiájának különbségeként jelenik meg). Ennek alapján az algoritmus a következő képlet szerint dolgozik:

$$S[t+1] = -\sum_{i=1}^{l} \left(\frac{S[t] - S_i[t]}{2} \cdot w_i\right) - D_0$$

A fenti képletben az S és az S_i jelentik az él két végén elhelyezkedő csomópontok energia állapotváltozóinak értékét, a w_i az él súlya, az l a jelenlegi csúcs fokszáma és a D_0 pedig az elszivárgási (disszipációs) tényező.

2.4.2 Egyéb modellek

A következő modelleket is támogatja a Turbine, viszont mivel ezek CUDA-ra történő átültetését jelen dolgozat nem tárgyalja, ezért csak említés szintjén kerülnek bemutatásra.

- *Gossip:*
  Ez a modell az adott csomópontokból történő információterjedést vizsgálja egy disszipációs tényező figyelembevételével.
- *Ripple:*
  Az energia hullámszerű terjedését szimulálja, szintén elszivárgási tényezővel együtt.
- *XY:*
  A játékelméleti, szinkron ismételt fogolydilemma szimulálását végzi ez a modell.

3 A VIDEOKÁRTYÁK FELÉPÍTÉSE

3.1 A GPU logikai felépítése CUDA szemszögből

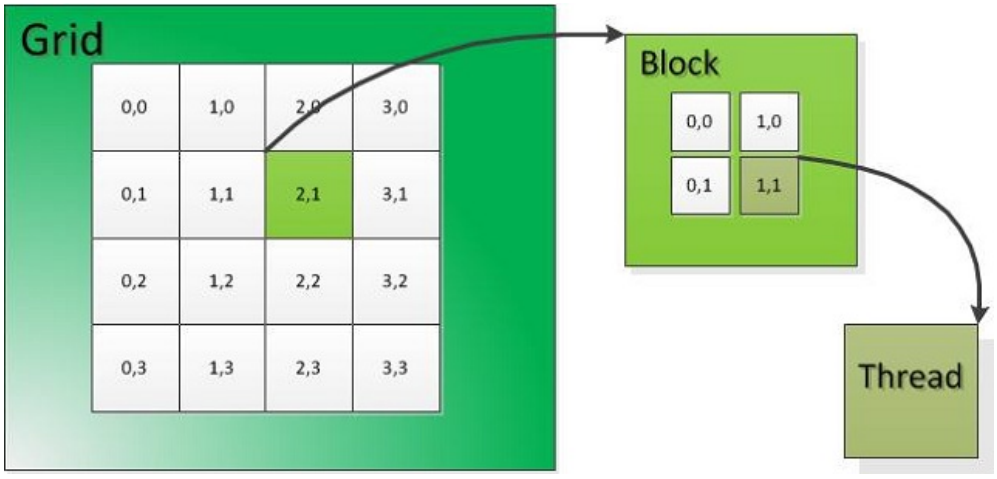
Egy CUDA program alapvető működése rengeteg sok szálon, párhuzamosan végrehajtott egyszerű utasításokból áll. Egy mai átlagos CPU körülbelül 4-8-, míg ezzel szemben egy GPU 25 000 szálat képes egyszerre futtatni[1], így érdemes az összetett feladatunkat nagyon sok egyszerű, párhuzamos részfeladatra felbontani.

Ezen rengeteg szálat logikailag három halmazba lehet csoportosítani, melyeknek mind megvannak a fizikai megfelelőik is.

A legkisebb elemi egység a *szál*. Mindegyik egy-egy példányban futtatja a GPU-n futó programot (az ún. *kernelt*), és saját, teljesen egyedi azonosítóval (szálindexszel) rendelkezik.

A *blokk* a szálak egy csoportja. A méretét bizonyos keretek között mi határozzuk meg. Egy blokkon belül létezik gyors kommunikáció az ún. megosztott memórián (*shared memory*) keresztül, illetve a szálakat utasíthatjuk, hogy egy blokkon belül egy adott utasítást elérve várják be egymást. De ez a lehetőség a blokkok között már nem áll fenn.

A legnagyobb csoport pedig a *grid*. Ez a blokkok csoportja; a méretét szintén meghatározhatjuk.



3.1. ábra: A grid, blokk és a szál kapcsolata

Itt érdemes még megemlíteni a *warp* fogalmát. A warp egy olyan logikai szálcsoportosítás, amelynek nincs fizikai megfelelője. A lényege az irányítás- és elágazásvezérlésnél jön elő: a warpban kötelezően összefogott 32 szál pontosan ugyanazt az utasítássorrendet követi egyszerre. Azaz amennyiben egy feltétel kiértékelésénél csak a warp egyik felére teljesül a feltétel igaz ága, akkor a warp másik fele addig várakozik, amíg az igaz ág végre nem hajtódik, majd az első tizenhat szál vár arra,

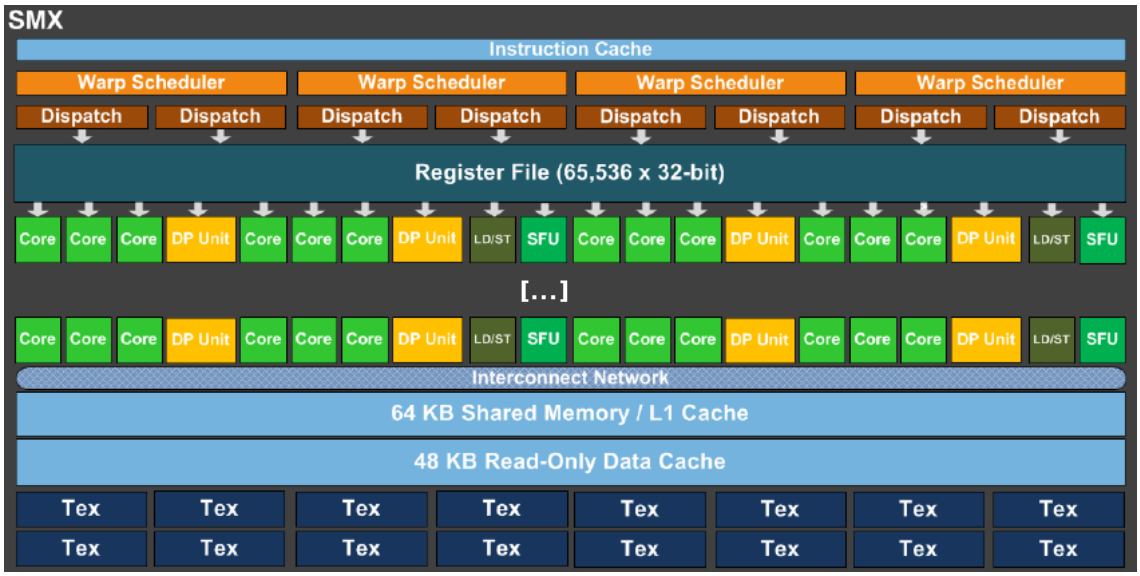
1 Egy NVIDIA GeForce GTX 580 esetében [14]: 16 multiprocesszor képes egyszerre 48 warpot futtatni, ahol egy warpban 32 szál fut. Így összesen ez a GPU 24 576 szálat képes egy időben kezelni.

hogy a hamis ág is végrehajtódjon, és utána újra együtt folyhasson a programvégrehajtás.

3.2 A GPU fizikai felépítése

A GPU felépítése SIMD jellegű (*Single instruction, multiple data*), azaz nagy adatmennyiségen végzik el a szálak ugyanazt a műveletet.

Ebből következően két fő komponenst kell megkülönböztetnünk: a globális adattároló memóriát, illetve az ún. streaming multiprocesszorokat (*Streaming Multiprocessor – SMX,* korábbi nevén *SM* vagy *MP*). Az előbbiről bővebben a 3.3. fejezetben bővebben esik szó, de elöljáróban annyit, hogy a felépítése analóg a CPU mellett lévő RAM-mal. Az SMX-ek felelősek a számításokért, belőlük jelenleg maximum 15 lehet egy kártyán.



3.2. ábra: A Streaming Multiprocessor (SMX) felépítése

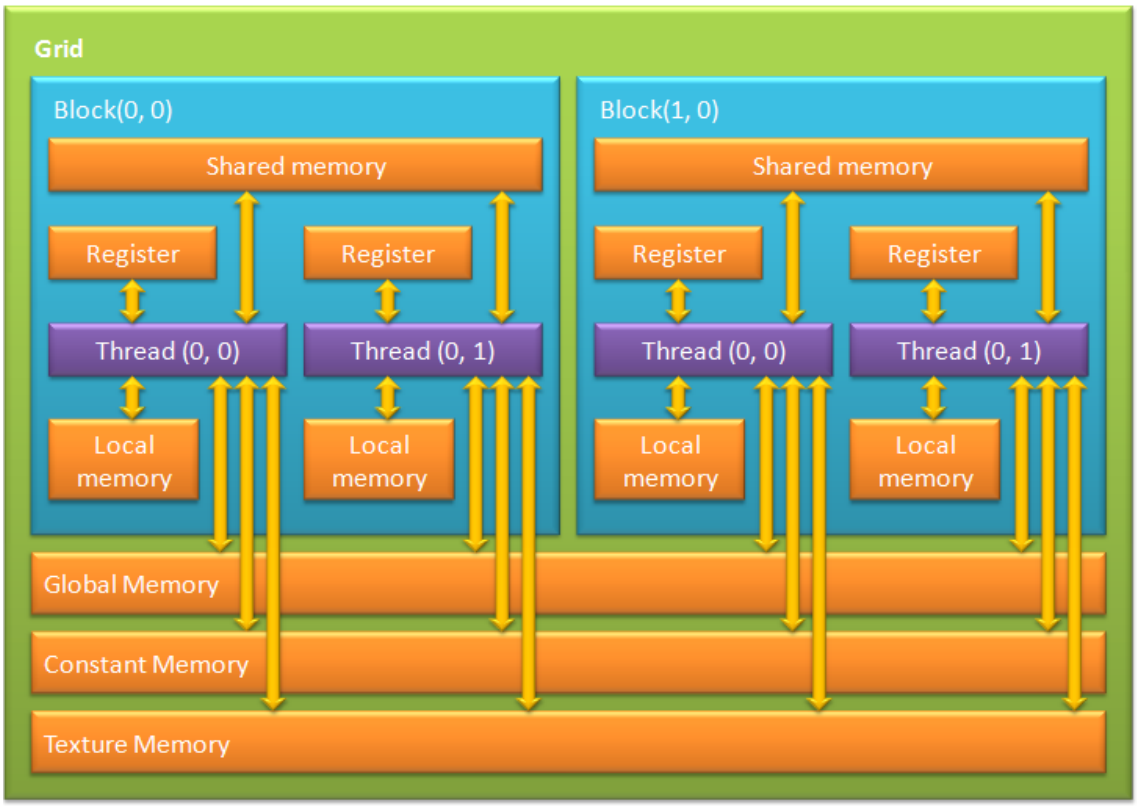
A 3.2. ábrán egy GK110 kártya SMX-ének felépítése látható. Minden ilyen tartalmaz négy warp ütemezőt, 192 egyszeres pontosságú- és 64 dupla pontosságú CUDA magot, itt nem tárgyalt speciális funkció egységeket (SFU) és load/store egységeket (LD/ST), illetve a 3.3. fejezetben bővebben ismertetett regisztereket, csak olvasható-, megosztott- és textúramemóriát is [15].

A kernelek CPU oldalról való hívásakor fontos lesz, hogy a korábban említett logikai csoportosítást, a grideket, blokkokat és szálakat a videokártyán lévő fizikai komponensekhez tudjuk kötni.

Ezen megfeleltetés egyszerű: ahogyan a 3.3. ábrán látszik, minden szál egy CUDA magon fut. Egy vagy több blokk egy SMX-en belül létezik, soha nincsenek szétosztva közöttük; a grid pedig a kártyán lévő GPU chipen fut [16].

3.3 Memóriatípusok

A memóriakezelésben vonhatunk némi párhuzamot a CPU-val: ugyanis minél közelebb kerülünk az adott szálhoz, annál kisebb, de gyorsabb memóriával is gazdálkodhatunk.



3.3. ábra: A különböző típusú memóriák elhelyezkedése a GPU-n

A 3.3. ábrán látható módon egy NVIDIA GPU-n öt memóriatípust érünk el [17]:

- Globális memória: ezek a videokártyán fizikailag jelen lévő SDRAM chipek. A programkódban bárhonnan hozzáférhető, módosítható, de mind közül a leglassabb. Mérete ma 1-12 GB között változik.
- Megosztott memória: fizikailag egy SMX-hez kapcsolt memóriaként jelenik meg (rajzon l. a 3.2. ábrát). A programban meg tudjuk benne osztani az adatokat, de csak a blokkon belül; itt bármelyik szál számára módosítható és jóval gyorsabb mint a globális memória. Mérete blokkonként 16-48 KB.
- Regiszterek: csak az adott szál változóit tudjuk benne kezelni. A leggyorsabb, módosítható, de teljesen lokális. A mérete nem a szálakhoz van kötve, hanem a blokkokhoz: összesen 8000-64 000 regiszterrel gazdálkodhatunk.
- Textúra memória: SMX-hez kötött, gyors, csak olvasható memóriaterület, amit a globális memóriából lehet feltölteni. Mérete 6-8 KB.
- Konstans memória: csak olvasható memória, szintén SMX-hez kötve. Mérete 8 KB.

4 CUDA C ismertető

A GPU a Streaming Multiprocesszorok (SMX-ek) köré épült. Egy többszálú program szálak blokkjaira van felosztva, amik egymástól függetlenül hajtódnak végre.

A következő fejezetekben ezen szálakon futó programok indításáról és speciális tulajdonságairól lesz szó a C programozási nyelven bemutatva.

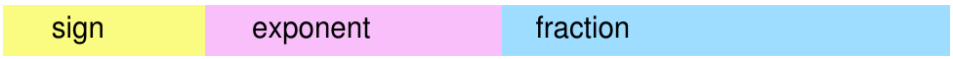
4.1 Kernel

A CUDA C nyelvi kiterjesztése lehetővé teszi, hogy a programozó olyan C függvényeket (ún. *kerneleket*) írjon, amelyek N-szer hajtódnak végre N különböző szálon.

Ehhez egy különleges módon, a „<<<...>>>” jelölővel kell meghívni az adott függvényt, majd az így, a videokártyán elindított program egy egyedi szálazonosítót kap.

4.2 Lebegőpontos számítási pontosság

Ebben a fejezetben szükséges megemlíteni a lebegőpontos számítások speciális kezelését és eltérését a CPU-s eredményektől, mivel a későbbiekben ez a formátum adja a számítások alapját.



4.1. ábra: A lebegőpontos ábrázolás felépítése az IEEE 754 szabvány szerint

Az IEEE 1985-ben elfogadta a bináris lebegőpontos aritmetikai szabványt (IEEE 754-1985 [18]), amit ezek után gyakorlatilag az összes számítástechnikai eszköz implementált, a CUDA architektúrát is beleértve. Ez a szabvány leírja, hogyan kell a lebegőpontos számtani eredményeket *közelíteni*.

2008-ban ezt kiegészítették [19] egy ún. *Fused Multiply & Add (FMA)* művelettel. Ezt használva csak egy kerekítési lépést vesz igénybe a műveletek végrehajtása, a legvégén, a változóba íráskor, így növelhető a pontosság. Ezt a kiegészítést a CUDA implementálta.

A helyzetet bonyolítja, hogy CPU oldalon az x87-es műveletek általában egy belső, pontosabb 80 bites lebegőpontos ábrázolást használnak, de nem alkalmaznak FMA-t, míg a szintén CPU oldalon jelen lévő SSE az IEEE szabvány szerinti 32 és 64 bit hosszú (float és double) ábrázolást alkalmazza, szintén FMA nélkül. Az x87 használata általában alapértelmezett a 32 bites fordítás esetén, míg az SSE a 64 bites fordítás esetén [20].

A fentiekből egyértelműen megállapítható, hogy a Turbine modelljének átalakítása után nem érdemes bitszinten összehasonlítani a double változókban kapott végeredményeket, hanem bizonyos kerekítési hibahatár mellett fogadjuk el őket.

2 A végrehajtó jelölő pontos szintaxisa: functionName<<<blocksPerGrid, threadsPerBlock>>>(arguments...), ahol a két, három dimenziós paraméterrel meghatározhatjuk az indítandó blokkok és a bennük lévő szálak számát.

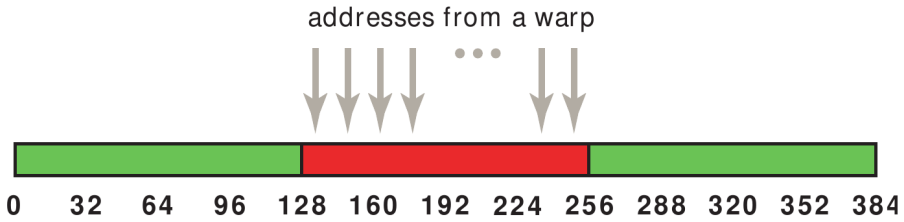
4.3 Olvasás a memóriából

A különféle struktúrák helyes felépítéséhez és gyors működéséhez szükséges megérteni, hogy a CUDA hogyan kezeli a memóriát.

Ahogyan láttuk a 3.3. fejezetben, a globális memória a leglassabban olvasható- és írható, viszont ez a legnagyobb is, így szinte minden adat, beleértve az egész hálózatot, abban fog tárolódni, így fontos, hogy tisztában legyünk az optimális kezelésével.

A globális memóriakezelés leglényegesebb alapelve az olvasások összefűzése, egyesítése (*coalesce*). Ezt a CUDA automatikusan elvégzi helyettünk warponként, ha bizonyos feltételek teljesülnek, ami pedig így összegezhető: egy warpban párhuzamosan futó minden szál egyidejű hozzáférései annyi műveletté egyesülnek, ahány gyorsítótár (*cache line*) szükséges ezek kiszolgálásához [21]. Compute Capability (CC) 2.x-szel rendelkező kártyák esetében az L1 cache van igénybe véve a globális memóriaolvasások eredményeinek tárolásához, ami 128 bájt méretű.

Ennek illusztrálásához tegyük fel, hogy a warpunk szálai négy bájt hosszú szavakat (pl. float-ot) olvasnak egymás után maximum 128 bájt hosszan. Ez látható a 4.2. ábrán:



4.2. ábra: Egyesített hozzáférés – minden szál egy gyorsítótárból olvas

Ha a pirossal jelölt terület valamelyik részét egy szál nem kérte volna, akkor is a cache-be kerülne, illetve ha ezek a kérések nem sorban, hanem véletlenszerűen történtek volna, de a jelölt területen belül maradnak, akkor is csak egy művelet maradt volna a globális memóriából olvasni.

Ha bármelyik kérés a zöld területbe esett volna, akkor azt a szegmenst is beolvasta volna a 128 bites L1 gyorsítótárba a CUDA, és így már két művelettel lehetne csak elérni a kért adatokat. Ezért fontos, hogy jól válasszuk meg a struktúráink felépítését, mert ha tegyük fel, csak minden második elemet dolgozzuk fel a memóriában egymás mellett ábrázolt adatokból, akkor ezen adatok betöltésének és eltárolásának hatékonysága (*load/store efficiency*) mindössze 50%-os lesz.

5 Átültetési megfontolások

A program tényleges videokártyára való átültetése előtt érdemes megvizsgálni, hogy milyen elméleti átültetési technikák léteznek, illetve milyen gyakorlati választási lehetőségek merülhetnek fel az átültetés közben.

5.1 Gustafson törvénye

Gustafson törvénye [22] (*weak scaling*) azt méri, hogy *processzoronként* hogy változik a megoldásig eltelt idő, ahogyan egyre több processzor kerül a rendszerbe, azaz amikor a teljes problémaméret úgy növekszik, ahogy a processzorok száma nő. Tehát a rendszer fejlesztésével a futási idő konstans marad.

Olyankor célszerű alkalmazni, amikor a problémaméret tud növekedni úgy, hogy kitöltse a rendelkezésre álló erőforrásokat (pl. folyadékok szimulálása vagy a Monte-Carlo-módszer futtatása, ahol a megnövelt erőforrások nagyobb pontosság elérését teszik lehetővé).

A Turbine maximális elérhető sebességnövekedését e törvény szerint fogjuk kiszámolni:

$$S = N + (1 - P) \cdot (1 - N)$$

Ahol P az összes soros végrehajtási idő azon része, amely párhuzamosítható; N pedig azon processzorok száma, ahol ez a kódrészlet fut.

5.2 Többdimenziós tömbök felépítése

C nyelvben tömbökről beszélünk, amikor ugyanazon típusú elemekből (esetünkben ez leggyakrabban a double) szeretnénk egy gyűjteményt létrehozni. Ekkor a memóriában ezen elemek egymás mellett helyezkednek el, és a tömbindexszel tudunk hivatkozni rájuk. Az, hogy hány darab indexre van szükség ahhoz, hogy egy elemet pontosan meghatározzunk, adja meg a tömb dimenziószámát (innen a többdimenziós elnevezés).

A C nyelv támogatja a többdimenziós tömbök létrehozását és kezelését, de a háttérben, a memóriában ezeket egydimenziósra lapítva kezeli.

A CUDA ezzel szemben sajnos nem ismeri a többdimenziós szintaktikai jelölést, így nekünk kell átgondolnunk és felépítenünk, hogyan fog megjelenni a C-ben egyszerűen megadható többdimenziós tömb egydimenziósra lapítva. Például egy *states[i+1][k]* tömb esetén egy eltolással kell kiváltani az i+1-et, pontosan úgy, ahogy a C végzi a címzést, mikor feloldja a dupla [ ]-rel elfedett szorzást: *states[(width * height) * (i + 1) + k]*.

5.3 Tömbök átvitele a videokártyára

A nem integrált típusú GPU és a CPU mellett elhelyezkedő memóriák fizikailag is elkülönülnek egymástól, így a processzor segítségével beolvasott és alakított adatokat (pl. a teljes hálózatot) mindenképpen szükséges átmásolni a videokártyára,

mielőtt dolgozni kezdenénk vele. Ebből következően a CPU oldalán lefoglalt memória címe nem lesz érvényes akkor, amikor a program a GPU-n fut és vice-versa[3].

A CUDA-ban is elérhetőek az eszközmemória lefoglalására és felszabadítására vonatkozó, a *malloc-hoz* és a *free-hez* hasonló függvények (a *cudaMalloc* és a *cudaFree*), amelyek csak az eszközön érvényes (csak ott dereferálható) mutatót adnak vissza. Ebből következően a dinamikus méretű tömbök struktúrabeli mutatóinak tárolását a gazda (CPU) oldalon egy köztes struktúrával kell megoldani.

Tegyük fel, hogy a Turbine hálózatunkban a csúcsokat és az éleket a következő tömbben szeretnénk tárolni:

```
struct cuda_network
{
    cuda_node *nodes;
    cuda_link *links;
};
```

Először a host oldalon létre kell hoznunk a cuda_network-öt, le kell foglalnunk a megfelelő mennyiségű csomópontot és élt, majd ezekbe belemásolni a Turbine-ban jelenleg natív formában lévő adatokat. Ezek után az eszközön is ugyanúgy le kell foglalnunk az azonos mennyiségű csúcsot és élt igénylő méretű struktúrát, majd a CPU memóriájából át kell másolni az adatokat a GPU memóriájába. Mindennek a tárolásához szükséges egy köztes struktúra; ehhez lásd a következő kódrészletet:

```
// Allocate space for the host network
cuda_network *h_network = new cuda_network;
h_network->nodes = new cuda_node[nodes_size];
h_network->links = new cuda_link[links_size];

// […] Convert and fill up the nodes and links array on the host

// Allocate space for the host-device connection network
cuda_network *d_h_network = new cuda_network;

// Allocate space for the device network on the GPU
cuda_network *d_network;
cudaMalloc(&d_network, sizeof(cuda_network))[4];
cudaMalloc(&d_h_network->nodes, nodesSize);
cudaMalloc(&d_h_network->links, linksSize);

// Copy the nodes and links from the host
```

3 Ez alól némi felmentést ad a *Unified Virtual Addressing (UVA)* nevű képesség, amely 2.0-s Compute Capability-től felfelé képes egy egységként láttatni a rendszer- és a videokártya memóriáját, de az alkalmazást nem teszi gyorsabbá, hiszen az adatmásolások megmaradnak [23]

4 A C-s változathoz hasonlóan a cudaMalloc memóriafoglaló függvény argumentumlistája: (mutató a lefoglalandó területre, ennek mérete)

```
cudaMemcpy(d_h_network->nodes, h_network->nodes, nodesSize)[5];
cudaMemcpy(d_h_network->links, h_network->links, linksSize);
cudaMemcpy(d_network, d_h_network, sizeof(cuda_network));
```

A fenti kódból már láthatóvá válik a köztes struktúra bevezetésének szükségessége: kell egy hálózatot tartalmazó cuda_network példány a host oldalon (*h_*), kell egy a device oldalon (*d_*), viszont mivel mindkettő mutatókat tartalmaz, ezért egyiket sem lehet a másik oldalról megcímezni. Viszont ahhoz, hogy a memóriamásolás működjön, két mutatóra van szükség, ezért létrehozunk egy köztes példányt (*d_h_*) a CPU oldalon, aminek a pointerei már a GPU oldalra hivatkoznak, majd végül a kitöltés után a d_h_network-öt átmásoljuk a d_network-be, így adva át a hoston lefoglalt GPU mutatók címeit a videokártyának.

6 Átültetés CUDA rendszerre

A Turbine szimulációjának alapvető felépítésének és a hozzá kapcsolódó fogalmak (2. fejezet) megismerése, illetve a CUDA felépítésének megvizsgálása (3. és 4. fejezet) után elkezdhetjük a program eddigi adatstruktúráinak videokártyára optimalizált változatainak elkészítését.

6.1 Élek

A vessels modell az élekkel dolgozik intenzíven, így ennek részletes a kidolgozása. Egy él a saját azonosítójával (*id*), a két végén elhelyezkedő csomópont azonosítójával (*node1_id* és *node2_id*), illetve az irányítottság (*directed*) tulajdonságával rendelkezik. Ennek megfelelően a *cuda_link* struktúra felépítése:

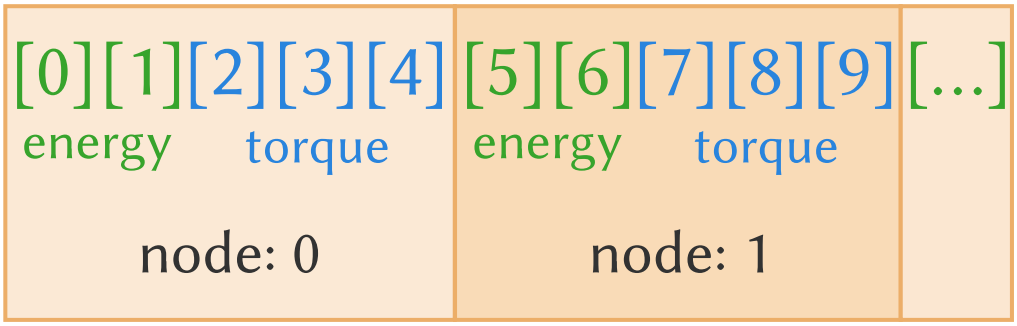
```
struct cuda_link
{
    unsigned long id;
    unsigned long node1_id, node2_id;
    char directed;
};
```

6.2 Állapotok

A Turbine-ban az állapotok azok a paraméterek, amik a szimuláció közben, annak hatására folyamatosan változnak. Kapcsolódhatnak egyaránt csomópontokhoz és élekhez is. Minden csúcsnak (és élnek) ugyanannyi állapota van. Szimuláció közben az adott modell igényelheti az előző vagy akár még az azt megelőző állapot értékét is az aktuális értékének kiszámításához.

Vagyük például azt az esetet, amikor a hálózatunk node-jai két állapottal rendelkeznek: energiával (*energy*) és nyomatékkal (*torque*). Az energia új értéke az előzőből számolódik, a nyomaték viszont az előző kettőből. Ebben az esetben az előbbihez két, az utóbbihoz három „belső állapot” tárolása szükséges.

5 A C-s változathoz hasonlóan a cudaMemcpy memóriamásoló függvény argumentumlistája: (hova, honnan, mennyit)



6.1. ábra: Az állapotok tárolására alkalmas tömb felépítése

Az összes állapotot tároló *states* double tömb egy dimenziós tömb (*flat array*), a benne lévő állapotelemek elhelyezkedését a 6.1. ábrán láthatjuk. Eszerint egy node-hoz hozzákapcsolódik a két állapot, „bennük" pedig a kettő, illetve három belső állapot egymás után. Ez blokk ismétlődik a hálózat minden egyes csúcsára.

A states tömb semmilyen olyan segédinformációval nem bír, mint a képen látható színezés vagy blokkhatárok, amivel a címzéshez szükséges indexeket ki lehetne számolni, így szükség van ezen segédtömbök létrehozására.

A legalapvetőbb információ, amit tudni kell, hogy egy elem (node vagy link) hány állapottal bír. Esetünkben ez kettő (az energia és a nyomaték). Ez a változó lesz a *state_per_element*.

Tudni kell továbbá, hogy mekkora puffert kell foglalni az egyes állapotoknak „belül"; jelen esetben az energiának kettőt, a nyomatéknak pedig hármat. Ez lesz a *buffer_per_state* tömb. Illetve a későbbi címzést gyorsabbá teszi, ha ezeket nem kell mindig összegezni, így egy cache változót is deklarálunk, a *buffer_size-t*.

Végezetül, minden szimulációs lépés lefutásánál tudnunk kell, hogy egy változó pufferének elemei közül jelenleg melyik az aktuális, amibe az eredményt írni kell, és melyek a korábbiak. Ennek tárolására szolgál a *current_read_position* tömb.

A legutóbbi két tömb mérete az állapotok számával egyezik meg.

A *cuda_state* struktúra végső kinézete tehát a következőképpen alakul:

```
struct cuda_state
{
    double *states;
    int state_per_element;
    int buffer_size;
    int *buffer_per_state;
    int *current_read_position;
};
```

Egy állapot (például a nyomaték utolsó pufferének) kiolvasása ezek alapján így zajlik: minden egyes szálhoz (kernelhez) hozzárendelünk egy csomópontot. A szál egyedi azonosítóval bír (*i*), ez segít a tömbcímzésben. A jelenlegi olvasási pozíció a nulla, ehhez képest kell még kettővel eltolnunk a címzést. Tudjuk, hogy a csúcsok egymás után következnek, és tudjuk, hogy egy csúcs öt helyet foglal el ebben a

tömbben (ez a puffer mérete). Ezek alapján így tudjuk párhuzamosan kiolvasni az összes node nyomatékának utolsó elemét:

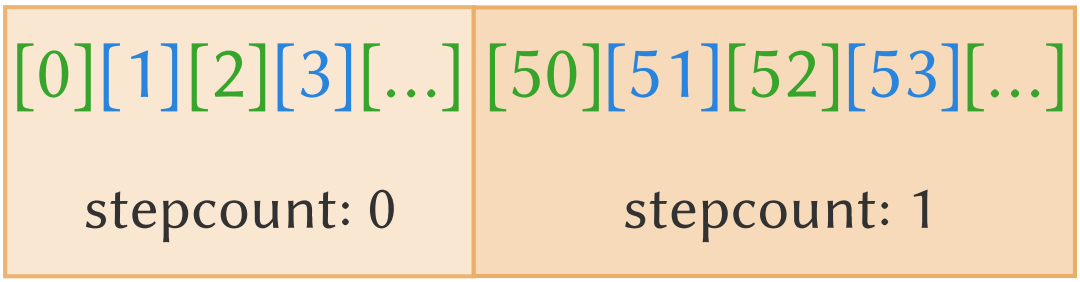
```
const int i = blockDim.x * blockIdx.x + threadIdx.x;
const int thread_offset = i * buffer_size;
const int offset = buffer_per_state[1];
const int read_position = current_read_position[1] + 2;
double value = states[thread_offset + offset + read_position];
```

Ha a jelenlegi olvasási helyzet nem nulla lenne, akkor könnyen a következő blokk értékét olvashatnánk, erre a valódi kódban figyelni kell.

6.3 Állapotmentések

A szimuláció minden lépésében kiszámolódnak a perturbációkkal módosított állapoteredmények, viszont ezeket a Turbine később vissza is tudja játszani, így szükséges a tárolásuk.

Nem az állapotstruktúrát kell egy az egyben lemásolni, hiszen a visszajátszáskor annál jóval kevesebb információra van szükségünk: egészen pontosan csak arra, hogy a jelenlegi állapotok olvasási pozícióiban milyen eredmények találhatóak. Ezt fogja tárolni a *history* tömb minden él vagy csúcs esetében a 6.2. ábrán látható módon.



6.2. ábra: A history tömb ábrázolja, hogyan tárolódnak az egyes szimulációs lépések után az állapoteredmények

A tömb felépítése a következő: rögtön a jelenlegi szimulációs lépés lefutása után egy ciklus bemásolja az összes node-hoz (és linkhez) tartozó aktuális olvasási pozícióban lévő állapotot (az ábrán most a zölddel jelölt energia és a kékkel jelölt nyomaték váltja egymást 50 csúcs erejéig). Majd a következő szimulációs ciklus után ez újra ismétlődik (50-től számozva).

Ez a tömb már fel van készítve arra, hogy a modellek esetleg megváltoztatják a csúcsok vagy élek számát, ezért azt is tárolni kell, hogy egy blokkban hány ilyen elem állapota foglal helyet. Ez a változó az *element_size* tömb, aminek nagysága a lépésszámmal (*stepcount*) egyenlő.

Ahhoz hogy tudjuk, hogy az utolsó elemet pontosan hova írtuk a tömbben, hogy a következő lépésben egyszerűen onnan tudjuk továbbfolytatni, vagy összeadjuk az összes eddigi blokk nagyságát vagy pedig eltároljuk ennek a helyét. E segéd neve az *end* változó.

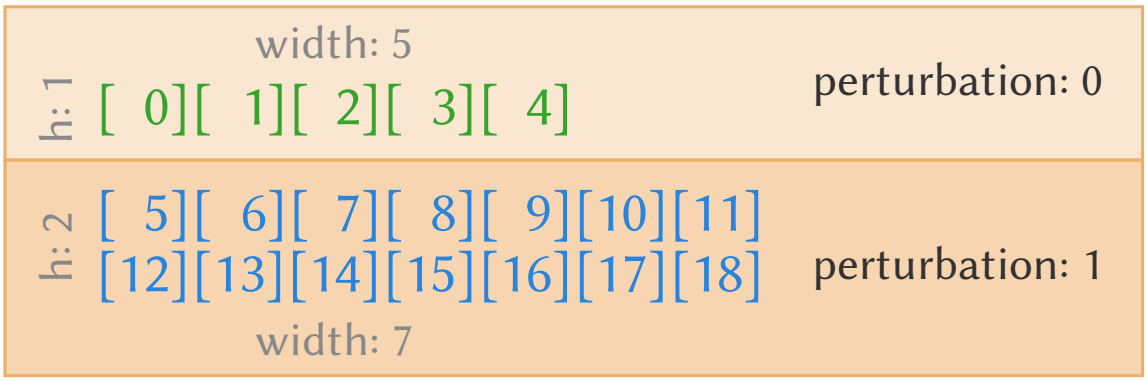
A fentiek alapján így néz ki a *cuda_state_snapshot* struktúra:

```
struct cuda_state_snapshot
{
    double *history;
    int *element_size;
    int end;
};
```

6.4 Perturbációk

Minden egyes állapothoz tartozik egy perturbációs függvény (pontosabban ennek kiszámolt értékei egy feltöltött tömbben). Egy perturbáció szélessége (*width*) közvetlen kapcsolatban áll az adott indexű gráfelem (csúcs vagy él) állapotával, erre az állapotra fog alkalmazódni (hozzáadódni vagy lecserélődni) a perturbációs érték. A magasság (*height*) pedig megfelel a lépésszámnak, tehát az első sorban lévő perturbációk csak az első szimulációs lépésben lesznek érvényesek, a második sorban lévők csak a második lépésben stb. Ez memóriatakarékossági okokból lehet ciklikus is.

A 6.3. ábrán látható a fenti példa szerinti két állapot (energia és nyomaték) perturbációinak ábrázolása rendre az első öt, illetve az első hét csomópontra.



6.3. ábra: Az energia és a nyomaték állapotokra vonatkozó perturbációk öt és hét csomópontra

A már ismert problémával kerülünk ismét szembe: a fenti egy dimenzióssá lapított tömb nem rendelkezik a határait leíró adatszerkezettel, így ezeket nekünk kell létrehozni.

Szükséges egy segédtömb a szélességeknek (*width*), egy a magasságoknak (*height*), egy a fentebb említett tulajdonságoknak (pl. hogy ciklikus-e az adott perturbáció) (*properties*), illetve egy a gyorsításért felelős, ami megmondja, hogy az adott perturbáció hol kezdődik az 1D-s tömbben, és így nem kell végigszámolni azt (*start_index*).

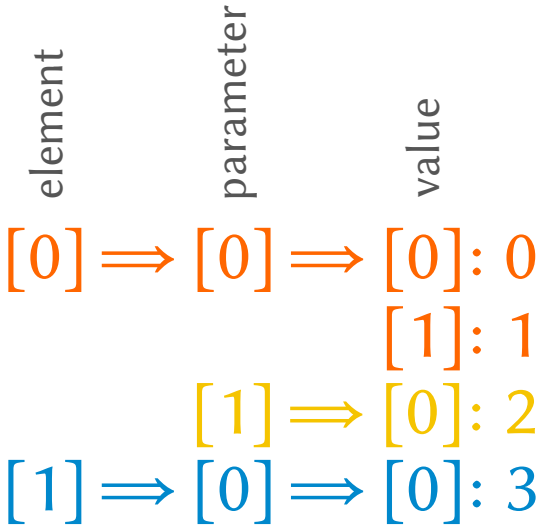
Ezek alapján a *cuda_perturbation* struktúra a következőképpen épül fel:

```
struct cuda_perturbation
```

```
{
    double *data;
    int *width;
    int *height;
    int *start_index;
    uint8_t *properties;
};
```

6.5 Paraméterek

Paraméterei mind a gráfelemeknek (csúcsoknak és éleknek), mind pedig a modelleknek lehetnek. Minden elem (*element*) egy vagy több paraméterrel (*parameter*) rendelkezhet és minden paraméter szintén egy vagy több értékkel (*value*) bírhat.



6.4. ábra: Több elemnek lehet több paramétere és azoknak több értéke

Ennek a „fizikai” ábrázolása meglehetősen könnyű, hiszen csak a paraméterértékeket kell egymás után írni a tömbbe. A kihívást ezen határok megtalálása- és az erre szolgáló segédtömbök megírása és kezelése jelenti. Ahogyan a 6.4. ábrán lévő példában láthatjuk, az első elem első paraméterének első két értéke a 0 és az 1, illetve második paraméterének értéke a 2 (sárga árnyalatokkal jelölve), míg a második elem első paraméterének első értéke a 3. Ezt egymás után ábrázolva a *data* tömbben a 0-1-2-3 értékeket kapjuk, ahogyan a 6.5. ábrán láthatjuk.

Két kiegészítő tömbre lesz szükségünk, hogy tudjuk, hogy melyik adat hova tartozik.

6.5. ábra: Az első négy paraméterérték tárolása és címzése

A *parameter_indices* tömb egy-egy indexet tárol minden egyes új paraméterre, míg az *element_indices* ugyanezt teszi az elemekkel. Ahhoz, hogy tudjuk, hogy az adott paraméter vagy elem olvasásának végéhez értünk, az utóbbi két tömbben mindig eggyel előrébb kell olvasni, és meg kell nézni, hogy a következő elem hol kezdődik. Ha például az adattömböt sorban olvassuk és a következő elem a 3-as, akkor megnézzük a paraméterindex tömböt, hogy a mostani adattömb hármasa már új paraméterhez tartozik-e.

Mivel a két segédtömböt mindig eggyel tovább olvassuk, ezért szükséges eggyel megnövelni a méretüket, hogy a legutolsó adatelemnél se címezzünk ki belőlük, így a végén az őrszem (*sentinel*).

Adminisztrációs szempontból érdemes még jegyezni ennek a három tömbnek a méretét is. Ennélfogva a *cuda_parameter* struktúra a következőképpen néz ki:

```
struct cuda_parameter
{
    double *data;
    int *element_indices;
    int *param_indices;

    int data_size, element_indices_size, param_indices_size;
};
```

6.6 A hálózati struktúra

A fentebbi fejezetekben részletesen volt szó a hálózati elemek felépítéséről, amelyeket néhány adminisztrációs változóval kiegészítünk és eggyé fogunk össze.

Az egységes hálózati struktúrának szükséges eleme még a csomópontok és élek számának rögzítése (*num_nodes* és *num_links*), az eddigi lépések száma (*stepcount*), a lépések növelésének mennyisége (*steptime*), illetve az a maximális lépésszám, amit a szimuláció futtatására fordítunk (*analysistime*). Ezeken kívül a modellek képesek megállapítani, ha egy hálózat stabilizálódott, így ennek tárolására is szükséges egy változó (*stabilised*).

Ezek alapján a *cuda_network* ernyőstruktúra a következőképpen néz ki:

```
struct cuda_network
{
    cuda_node *nodes;
    cuda_link *links;
    cuda_state node_states, link_states;
    cuda_state_snapshot node_state_snapshot, link_state_snapshot;
    cuda_parameter node_parameters, link_parameters, model_parameters;
    cuda_perturbation node_perturbations, link_perturbations;

    int num_nodes, num_links;
```

```
    double steptime;
    int analysistime;
    unsigned long stepcount;
    bool stabilised;
};
```

6.7 A vessels modell

A fenti struktúrák megtervezése után a vessels modell CUDA változatának átírása jelenti az utolsó lépést a szimuláció indításához. Ahogyan a 2.4.1. fejezetben látható volt, a modell a következő képlet alapján dolgozik:

$$S[t+1] = -\sum_{i=1}^{l} \left(\frac{S[t] - S_i[t]}{2} \cdot w_i \right) - D_0$$

Ebben a képletben a két S az él két végén elhelyezkedő csomópont energiáját takarja, a w az él áteresztőképességének felel meg, míg a D_0 az elszivárgás mértéke.

Ennek alapján és a fenti struktúra fényében a legegyszerűbb dolgunk akkor van, ha az összes élen végiglépkedünk, majd kiszámoljuk a jelenlegi időpillanatban (t) érvényes, az él két oldalán elhelyezkedő csúcs energiáját (ennek az összegnek lesz a neve a *carry*), majd pedig ezt hozzáadjuk a következő lépés állapotához ($t+1$).

A megoldás legnagyobb előnye, hogy az összes élen végiglépdelni egyszerre, párhuzamosan is lehet, így itt mutatkozik meg a videokártyás gyorsítás hatalmas előnye. Viszont abba is bele kell gondolni, hogy mi történik akkor, ha egy csomóponthoz több link is fut és azokat egyszerre dolgozzuk fel. Ekkor több szál próbálja ugyanabban a pillanatban az előző lépésből már létező energia értékéhez hozzáadni a saját maga által kiszámolt carry-t, viszont így csak az egyik jut érvényre (az, amelyik utoljára ír a memóriaterületre). Ezért szükséges bevezetni az atomi műveleteket, amik akkor térnek vissza, ha a terület sikeresen módosult az általunk elvártra. Ehhez egy CAS (*compare-and-swap*) műveletet használó *atomicAdd* függvény definiálása szükséges, ami egészen addig próbálkozik a régi érték kiolvasásával és a carry hozzáadásával, amíg sikerrel nem jár. Sajnos ez teljesítményvesztéssel jár, de ebben az esetben csak így garantálható a helyesség.

Ennek a kódja CUDA nyelven:

```
__device__ double atomicAdd(double *address, const double val)
{
    unsigned long long int* address_as_ull = (unsigned long long int*)
                                             address;
    unsigned long long int old = *address_as_ull, assumed;

    do {
        assumed = old;
        old = atomicCAS(address_as_ull, assumed,
            __double_as_longlong(val + __longlong_as_double(assumed)));
```

```
    } while (assumed != old);

    return __longlong_as_double(old);
}
```

Tudjuk, hogy jelen esetben az állapottárolónk puffere (l. 6.2. fejezet) két állapot hosszú: az egyik tárolja a jelenlegi energiát (t), míg a másik a következő lépés energiáját ($t+1$). Ennek fényében olvasni a *current_read_position* változóból szükséges, írni pedig a „másikba”, amit egy egyszerű negálással kaphatunk meg (*inv_current_read_position*).

A fentiek szerint a linkeken való végiglépdelés (az i szálazonosító jelöli az adott él azonosítóját) a következő kernellel valósítható meg:

```
__global__ void vessels_links_kernel(cuda_network *d_network)
{
    const int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i >= d_network->num_links) {
        return;
    }

    const int current_read_position = d_network->
                                node_states.current_read_position[0];
    const int inv_current_read_position = !current_read_position;

    const double steptime = d_network->steptime;
    const unsigned long node1_id = d_network->links[i].node1_id;
    const unsigned long node2_id = d_network->links[i].node2_id;

    // carry = strength * (state[node1] - state[node2]) / 2 * steptime
    const double carry = d_network->link_parameters.data[i] *
(d_network->node_states.states[(2 * node1_id) + current_read_position]
- d_network->node_states.states[(2 * node2_id) +
current_read_position]) / 2 * steptime;

    atomicAdd(&(d_network->node_states.states[(2 * node1_id) +
inv_current_read_position]), -carry);
    atomicAdd(&(d_network->node_states.states[(2 * node2_id) +
inv_current_read_position]), carry);
}
```

Észrevehető, hogy a fenti kódban nem szerepel sem az elszivárgás számítása, sem pedig a stabilitásvizsgálat. Ez egy új, az előző lefutása után induló kernellel valósítható meg, ami minden node (szintén i jelöli az azonosítót) állapotán megy végig, és számítja ki az említett két tulajdonságot.

A disszipciót (az első modellparamétert) az energia eredménypufferéből csak ki kell vonni, így ennek a megírása elég egyszerű:

```
d_network->node_states.states[(2 * i) + inv_current_read_position] -=
d_network->model_parameters.data[0] * d_network->steptime;
```

Az aktuális kódban továbbá figyelni kell arra, hogy nulla alá ne csökkenjen az energia.

Egy hálózat akkor stabilizálódott a vessels modell esetében, ha minden egyes csúcsának energiája nulla. Ezt a következőképpen lehet megállapítani: minden szimulációs lépés előtt a hálózat *stabilised* állapotváltozóját (l. 6.6. fejezet) igazra állítjuk, majd ha találunk egy nem nulla értéket, akkor ezt hamisra állítjuk vissza. Az *__any* CUDA függvény visszatérési értéke igaz, hogy az egy warpon belül futó bármelyik szálnál a paraméterként adott elem értéke nem nulla. Ez a függvény meggátolja a végrehajtás elágazását a warpon belül, így gyorsító hatással bír (l. 3.1. fejezet).

Ennek megfelelően a hálózat stabilitását vizsgáló kód:

```
if (d_network->stabilised && __any(d_network->node_states.states[
                               (2 * i) + inv_current_read_position])
   )
{
    d_network->stabilised = false;
}
```

6.8 A CUDA szimuláció menete

A Turbine a szimulációt a különféle adatfájlok (l. 2.2. fejezet) beolvasásával kezdi, majd betölti a modelleket és a hálózatot. Ezek után lefuttatja a szimulációt és a megváltozott hálózatot és az új állapotokat a lemezre írja.

A CUDA ebben a folyamatban a szimulációban vesz részt. Egészen pontosan a következőket kell megtenni ahhoz, hogy a Turbine formátumában beolvasott hálózat a videokártyán is szimulálható legyen:

1. Be kell tölteni a hálózatba a kiinduló állapotokat. Ez még a CPU oldalon történik meg, mert mindössze egy pár tizedmásodperces folyamatról van szó.
2. Ezek után a lemezről beolvasott és az előbbi pontban előkészített hálózatot át kell alakítani a fentebbi alfejezetekben tárgyalt CUDA hálózattá. Ez annyit tesz a gyakorlatban, hogy minden egyes hálózati elemet (mint a linkek, állapotok, perturbációk stb.) kiolvasunk a Turbine szintaktikája szerint és a már lefoglalt CUDA hálózati tömbökbe írjuk be azokat.
3. Ezen lépés után a CPU mellett lévő memóriában jelenleg két hálózat tartózkodik. A cuda_network szerinti felépítésűt át kell másolni a videokártyára, miután lefoglaltuk a helyet az eszközön.
4. Ezek után a meghatározott lépésszámszor el kell indítani a szimulációt, aminek a CUDA-ra átírt változata minden egyes alkalommal módosítja a GPU-n lévő hálózatot:

4.1. Elsőként az adott lépéshez tartozó perturbációt kell alkalmazni a hálózatra (l. 6.4. fejezet).

4.2. Ezek után az adott modell (esetünkben a vessels) a benne meghatározott algoritmus szerint módosítja az állapotértékeket, illetve akár a csúcsok és élek számát is.

4.3. Ekkor már rendelkezünk a jelenlegi szimulációs lépés végső állapotváltozóival, így el lehet menteni azokat egy state_snapshot tömbbe (l. 6.3. fejezet).

4.4. A mentés után meg kell vizsgálni, hogy a hálózat stabilizálódott-e (vagy elérte-e a végső lépésszámot), és ha igen, akkor kilépünk a ciklusból, egyéb esetben pedig növelünk egyet a lépésszámon és folytatjuk a 4.1-es ponttól.

5. Ha véget ért a szimuláció, akkor vissza kell másolni a módosított adatokat a host memóriába, majd pedig fel kell szabadítani a videokártyán a lefoglalt helyet.

6. A visszamásolt, friss adatokkal feltöltött cuda_network állapotváltozóit vissza kell írni a Turbine típusú hálózatba, végül pedig törölni szükséges a tömb által lefoglalt helyet.

7 Teljesítménymérés

7.1 Elméleti maximális teljesítménynövekedés

Ahogyan az 5.1. fejezetben láthattuk, Gustafson törvénye alapján érdemes felállítani azt az elméleti maximumot, amit a kód gyorsításával elérhetünk. A képlet a következő volt:

$$S = N + (1 - P) \cdot (1 - N)$$

Ahol P az összes soros végrehajtási idő azon része, amely párhuzamosítható; N pedig azon processzorok száma, ahol ez a kódrészlet fut.

Ezek alapján meg kell vizsgálnunk a CPU-n futó Turbine esetében, hogy mekkora a jól párhuzamosítható kódrészletek száma. Ezt a Callgrind függvényhívási gráf generátor eszközzel tudjuk megtenni, ami a Valgrind programcsomag része [24]. Az eszköz lefuttatása után azt az eredményt kapjuk, hogy a vessels modellben, a linkeken való végiglépdelés viszi el a futási idő 87%-át. Ha ezt a ciklust teljes mértékben párhuzamossá tudjuk tenni a tesztgépben elhelyezkedő 960 CUDA magon (a fizikai felépítésről l. a 3.2. fejezetet), akkor elméletben, a fenti képlet szerint, maximálisan 835-szörös gyorsulást tapasztalhatunk. Ez az érték tényleg az elméleti maximum, nem veszi figyelembe a hardverkorlátokat, az utasításon belüli elágazások miatti lassításokat, (l. 3.1. fejezet), a memóriaírások- és olvasások idejét (l. 3.3. fejezet), ezért a gyakorlatban a 70-100-szoros gyorsulási érték már nagyon jónak számít.

7.2 Teszthálózatok generálása

Ahogyan a 2.3. fejezetben láthattuk, a Turbine négy hálózattípust támogat, melyeknek mind más a felépítésük, így azonos mennyiségű csomópontokhoz más mennyiségű él csatlakozik. Ezek a hálózatok a program terminológiájában a *grown* (Barabási–Albert), a *lattice* (rácsháló), a *random* (Erdős–Rényi) és a *smallworld* (Watts–Strogatz).

Mivel a CUDA vessels szimuláció az éleken dolgozik elsősorban (l. 6.7. fejezet), ezért az a sejtés, hogy az azonos számú csomóponttal, de kevesebb éllel rendelkezdő hálózatokon rövidebb lesz a futásidő.

Ehhez mindegyik hálózatból el kell készíteni a 100, 1000, 10 000, 100 000 csomópontos változatot, amelyeknél mindhez tartozik egy-egy változat 100, 1000, 10 000, a 100 000, 1 000 000 és 10 000 000 éllel. Mivel élenként dolgozzuk fel az adatokat a modellben, ezért figyelni kellett rá, hogy legalább annyi vagy több él legyen a hálózatban mint csúcs, máskülönben az egyedülálló csúcsok csak a memóriát foglalnák. A négyzetes rácsháló alakú gráf (lattice) sajátossága miatt az adott számú csomópont már meghatározta a köztük lévő élek számát is, illetve smallworld hálózatban nem lehetett ugyanannyi élnek és csúcsnak lennie. Továbbá a random gráf élei nem pontosan „kerek számok” a véletlen eloszlásnak köszönhetően.

A hálózatok, az elkészítésük után, a csúcsaik felére véletlenszerűen egy 10 000 nagyságú Gauss eloszlású gerjesztést kaptak az első lépésben egy perturbáció formájában. Ez már elegendő volt ahhoz, hogy a kiterjedt hálózatok se stabilizálódjanak a szimuláció befejeződése előtt.

A fenti kritériumok alapján egy linuxos bash script összesen 35 hálózatot generált 1470 MB tárhelyet felhasználva.

7.3 Szimulációk futtatása

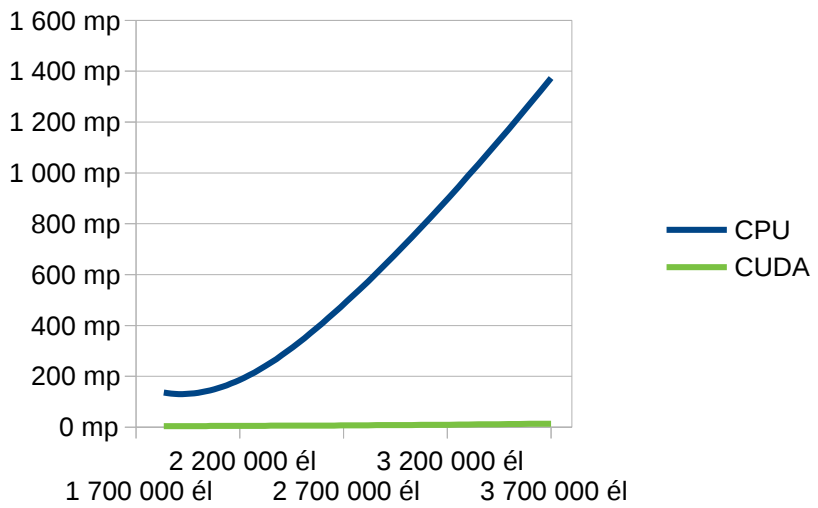
A fentebbi hálózatok mindegyikén legalább négy szimulációt kell futtatni a vessels modellel mind CPU-n, mind GPU-n., hogy biztosak lehessünk az eredményben. Ezek a programok egymáshoz képest sorosítva futnak, hogy ne vegyék el egymástól a futásidőt.

Összesen 466 darab, ezer lépéses szimuláció futott le körülbelül két nap alatt.

A futásidőben CPU és GPU esetén a perturbációk számolása, a vessels modell futtatása, az eredményállapotok mentése, illetve a stabilitásvizsgálat szerepel. Továbbá a GPU-n beleszámít az időbe a hálózat átalakítása cuda_network formátumúra, az adatok feltöltése a videokártyára és a letöltésük is onnan. Egyik helyen sem számít az eltelt időbe a hálózat merevlemezről történő beolvasása vagy az oda történő kiírása.

A CPU-n és a GPU-n kapott eredmények az első tizenkét tizedesjegyig megegyeznek, az utána történő eltérések a különbözőképpen megvalósított lebegőpontos számítások miatt lehetségesek (erről bővebben l. a 4.2. fejezetet).

A szimulációk futási időinek átlagai a 7.1–7.4. táblázatokban találhatók meg.



7.1. ábra: A grown network szimulációs ideje a csomópontok és az élek számának növekedésével

100 csomópont	99 él	990 él	9 900 él	99 000 él	990 000 él	9 900 000 él
CPU	0,01 mp	0,08 mp	0,82 mp	7,73 mp	74,72 mp	731,77 mp
CUDA	0,25 mp	0,24 mp	0,26 mp	0,45 mp	2,31 mp	19,98 mp
Gyorsulás	0,1×	0,3×	3,2×	17,4×	32,4×	36,6×

1 000 csomópont	999 él	9 990 él	99 900 él	999 000 él	9 990 000 él
CPU	0,17 mp	1,24 mp	10,47 mp	90,44 mp	865,85 mp
CUDA	0,25 mp	0,25 mp	0,47 mp	2,42 mp	21,46 mp
Gyorsulás	0,7×	5,0×	22,4×	37,3×	40,4×

10 000 csomópont	9 999 él	99 990 él	999 900 él	9 999 000 él
CPU	4,58 mp	26,65 mp	214,99 mp	1 919,25 mp
CUDA	0,53 mp	0,84 mp	2,88 mp	24,21 mp
Gyorsulás	8,7×	31,8×	74,8×	79,3×

100 000 csomópont	99 999 él	999 990 él	9 999 900 él
CPU	68,33 mp	398,43 mp	3 653,00 mp
CUDA	3,22 mp	5,64 mp	32,07 mp
Gyorsulás	21,2×	70,6×	113,9×

7.2. táblázat: A grown hálózat eredményei

100 csomópont	180 él
CPU	0,02 mp
CUDA	0,21 mp
Gyorsulás	0,1×

1 000 csomópont	1 860 él
CPU	0,19 mp
CUDA	0,21 mp
Gyorsulás	0,9×

10 000 csomópont	19 800 él
CPU	2,29 mp
CUDA	0,29 mp
Gyorsulás	8,0×

100 000 csomópont	199 080 él
CPU	41,85 mp
CUDA	2,37 mp
Gyorsulás	17,7×

7.1. táblázat: A lattice hálózat eredményei

1 000 csomópont	994 él	9 954 él	100 016 él	1 000 000 él
CPU	0,19 mp	1,67 mp	13,86 mp	85,69 mp
CUDA	0,23 mp	0,26 mp	0,50 mp	1,94 mp
Gyorsulás	0,8×	6,3×	27,7×	44,2×

10 000 csomópont	99 765 él	998 266 él	9 998 124 él
CPU	45,95 mp	384,92 mp	3 243,07 mp
CUDA	0,66 mp	2,91 mp	21,45 mp
Gyorsulás	0,7×	2,9×	21,5×

100 000 csomópont	9 999 194 él
CPU	4 427,03 mp
CUDA	47,25 mp
Gyorsulás	93,7×

7.3. táblázat: A random hálózat eredményei

1 000 csomópont	10 000 él	100 000 él
CPU	1,11 mp	11,32 mp
CUDA	0,26 mp	0,39 mp
Gyorsulás	4,3×	28,8×

10 000 csomópont	100 000 él	1 000 000 él
CPU	18,51 mp	143,94 mp
CUDA	0,48 mp	1,81 mp
Gyorsulás	38,6×	79,7×

100 000 csomópont	1 000 000 él
CPU	207,14 mp
CUDA	3,58 mp
Gyorsulás	57,9×

7.4. táblázat: A smallworld hálózat eredményei

Ahogyan a 7.1. ábrán (és a 7.2. táblázatban) nagyon szépen látható, minél inkább összetettebbé válik a hálózat, annál inkább érvényesül a videokártya ereje.

Érdemes megfigyelni azt is, hogy a csomópontok vagy az élek növekedését tolerálják-e inkább a szimulációk. Ehhez először minden hálózatnál meg kell vizsgálni a rajtuk futott szimulációk átlagát. Ezeket a 7.5–7.8. táblázatok tartalmazzák.

	100 csomópont 1 833 332 él	**1 000 csomópont 2 219 978 él**	**10 000 csomópont 2 777 222 él**	**100 000 csomópont 3 699 963 él**
CPU	135,86 mp	193,63 mp	541,36 mp	1 373,25 mp
CUDA	3,91 mp	4,97 mp	7,11 mp	13,64 mp
Csomópont növekedés		10,00	100,00	1 000,00
Él növekedés		1,21	1,51	2,02
CPU végrehajtási idő növekedése		1,43	3,98	10,11
CUDA végrehajtási idő növekedése		1,27	1,82	3,49

7.5. táblázat: A grown hálózat átlagos csomópontjai, élei és futásidejei, illetve az első szimulációhoz képesti növekedés

	100 csomópont 180 él	**1 000 csomópont 1 860 él**	**10 000 csomópont 19 800 él**	**100 000 csomópont 199 080 él**
CPU	0,02 mp	0,19 mp	2,29 mp	41,85 mp
CUDA	0,21 mp	0,21 mp	0,29 mp	2,37 mp
Csomópont növekedés		10,00	100,00	1 000,00
Él növekedés		10,33	110,00	1 106,00
CPU végrehajtási idő növekedése		9,50	114,50	2 092,50
CUDA végrehajtási idő növekedése		1,02	1,39	11,54

7.6. táblázat: A lattice hálózat átlagos csomópontjai, élei és futásidejei, illetve az első szimulációhoz képesti növekedés

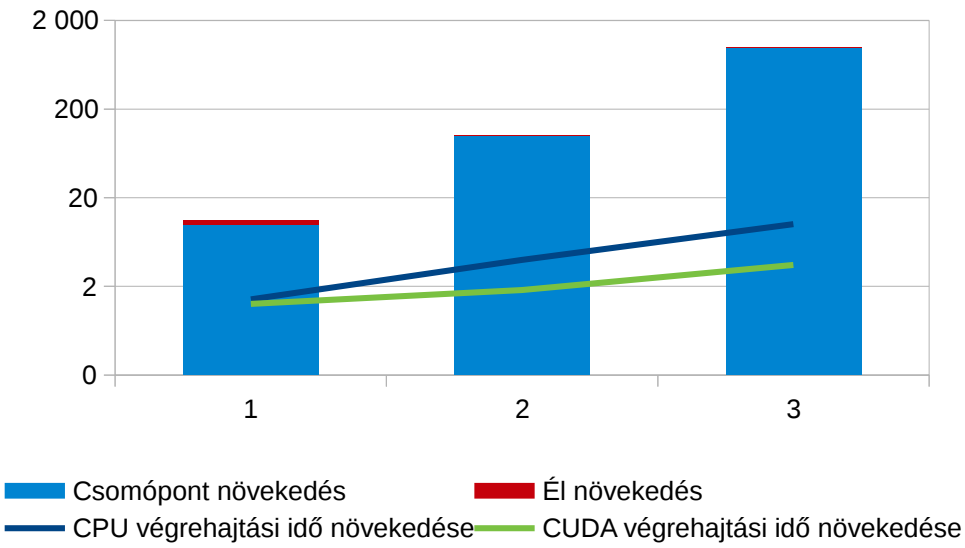
	1 000 csomópont 55 000 él	**10 000 csomópont 550 000 él**	**100 000 csomópont 1 000 000 él**
CPU	6,22 mp	81,23 mp	207,14 mp
CUDA	0,33 mp	1,14 mp	3,58 mp
Csomópont növekedés		10,00	100,00
Él növekedés		10,00	18,18
CPU végrehajtási idő növekedése		13,06	33,31
CUDA végrehajtási idő növekedése		3,50	10,95

7.7. táblázat: A smallworld hálózat átlagos csomópontjai, élei és futásidejei, illetve az első szimulációhoz képesti növekedés

	1 000 csomópont 277 741 él	10 000 csomópont 3 698 718 él	100 000 csomópont 9 999 194 él
CPU	25,35 mp	1 224,65 mp	4 427,03 mp
CUDA	0,73 mp	8,34 mp	47,25 mp
Csomópont növekedés		10,00	100,00
Él növekedés		13,32	36,00
CPU végrehajtási idő növekedése		48,31	174,64
CUDA végrehajtási idő növekedése		11,37	64,44

7.8. táblázat: A random hálózat átlagos csomópontjai, élei és futásidejei, illetve az első szimulációhoz képesti növekedés

A 7.5. táblázat alapján készült 7.2. ábrán látható, ahogyan minden egyes szimulációnál tízszeresére nő a csúcsok száma, míg az élek csak 1,5-2-szeresre. Viszont a várakozásoknak megfelelően, a szimulációk futásidejei inkább a linkekhez kötötten, jóval kisebb arányban nőnek, a CPU-éi gyorsabban, míg a GPU-éi lassabban.



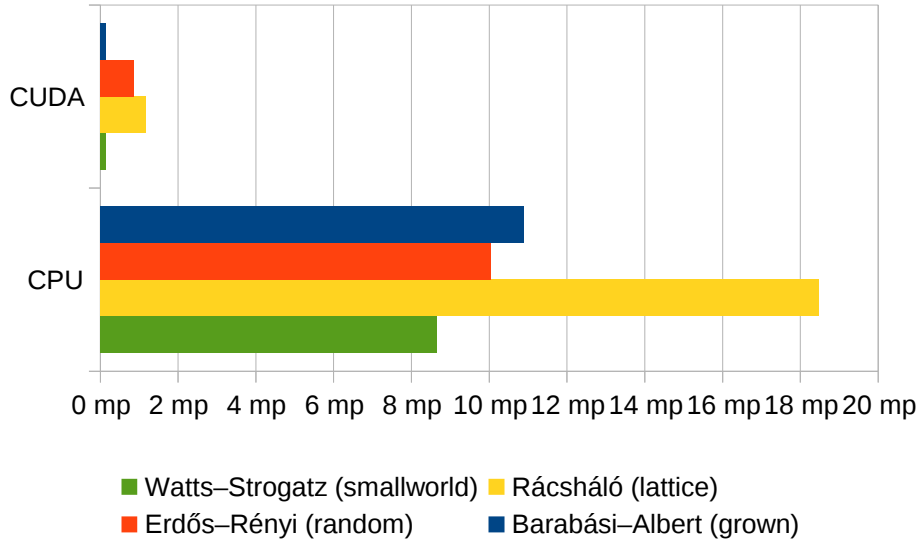
7.2. ábra: A grown hálózat csomópontjainak és éleinek növekedésének a szimulációs időre gyakorolt hatása. Logaritmikus skála.

Továbbá érdemes még megvizsgálni, hogy melyik típusú hálózaton mennyire teljesít jól a kétfajta felépítésű rendszer. Mivel láttuk, hogy a leginkább a linkek határozzák meg a futásidőt, ezért minden hálózatnál 50 000 él átlagát kiszámolva meg kell nézni a futásidőt. Ezt a 7.9. táblázat tartalmazza. Mivel a random és a small-world gráfokban nem szerepeltek 100 node-os hálózatok, így ezek az alább látható átlagokban sem szerepelnek a másik két gráfban sem.

	grown	lattice	random	smallworld
CPU	10,89 mp	10,04 mp	18,47 mp	8,64 mp
CUDA	0,14 mp	0,86 mp	1,17 mp	0,15 mp
Gyorsulás	77,0×	11,6×	15,8×	58,6×

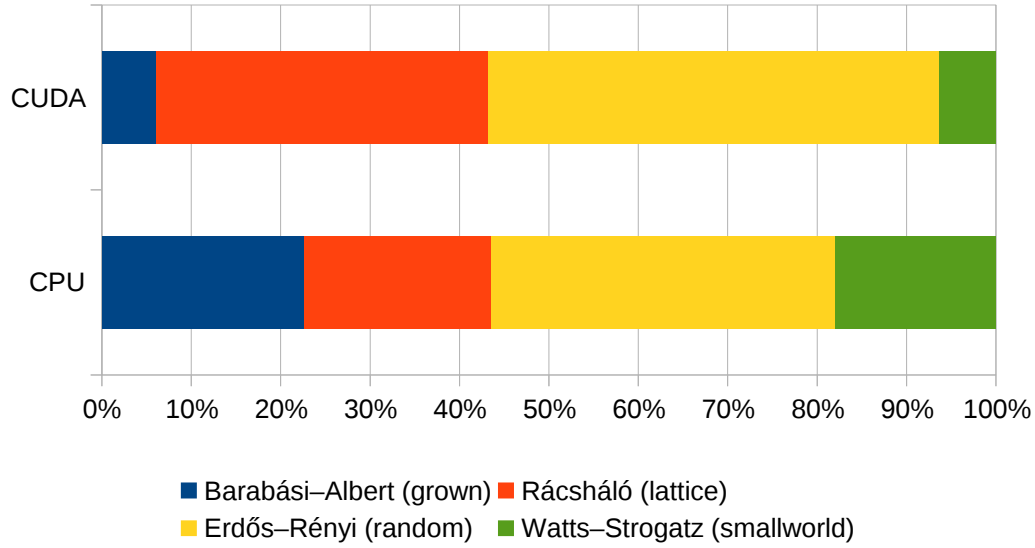
7.9. táblázat: 50 000 éllel rendelkező hálózatok szimulációjának átlagos futásideje

Ahogyan a fenti adatsoron (és vizualizálva a 7.3. ábrán) látható, nem egyforma mennyiségű futásidőt tölt a CPU és a GPU a különféle hálózatokkal. Az tisztán látszik, hogy a CUDA-s megoldás egyértelműen gyorsabb, de van, ahol csak 11-szer, míg van ahol már 77-szer.



7.3. ábra: 50 000 éllel rendelkező hálózatok átlagos futásideje GPU-n és CPU-n

Ahogyan a 7.4. ábrán megfigyelhető, azoknál a gráfoknál, minél inkább egyenlőtlenebb a fokszámeloszlás, annál inkább előnybe kerül a GPU-s megoldás a CPU-ssal szemben. Ennek az az oka, mint ahogyan a 6.7. fejezetben láttuk a vessels modell megvalósításánál, a csúcsokhoz tartozó állapotok írása atomi művelet, azaz egyszerre csak egy, az egyik élhez tartozó CUDA szál tudja írni az állapotot, a többinek addig várakoznia kell. Így sérül a nagyfokú párhuzamosság, mert az atomicAdd műveleteknél a kód egy nagyon rövid ideig soros végrehajtásúvá válik. Ebből pedig következik, hogy a hálózat méretéhez képest minél több él kapcsolódik egy csomóponthoz (mint ahogyan az történt a random és a lattice gráfoknál), annál inkább előtűnik az atomi műveletek végrehajtásának költsége.



7.4. ábra: Az átlagos futásidők százalékos eloszlása az egyes rendszerek között az 50 000 éllel rendelkező hálózatokon

Végül nézzük meg, hogy a fentebb létrehozott és szimulált gráfoknál mekkora átlagos gyorsulást sikerült elérnünk:

Csomópontok	31 729 csomópont
Élek	1 969 883 él
CPU	477,06 mp
CUDA	5,79 mp
Gyorsulás	82,5×

7.10. táblázat: A gyorsulás átlaga a CPU-s szimulációhoz képest

Mint az a 7.10. táblázatban látszik, a fentebb elvégzett szimulációk átlagosan 82-szeres gyorsulást mutattak a CPU-s változattal szemben , ha CUDA alatt futottak.

8 Összefoglalás

A dolgozatban megismertük a dinamikushálózat-analízist és azt, hogy az organikusan kialakult gráftopológiák hasonlítanak egymásra, ezért nem szükséges minden egyes alkalmazási terület vizsgálatára külön programot írni, érdemes a már meglévőeket fejleszteni.

Ez a program a Turbine, amely képes különféle, létező gráfokat beolvasni vagy akár újakat is generálni, és a bennük lévő élekhez és csúcsokhoz adatokat (állapotokat) rendelni, majd ezeken különféle modellek szerint vezérelt szimulációkat futtatni.

Megtudtuk, hogy ezek a szimulációk egymás után több százszor vagy több ezerszer futnak egy nagy kiterjedésű gráfon, ami ideálissá teszi a GPGPU-ra, videokártyára történő átültetéshez. Bemutatásra került a CUDA, mint ennek a programnyelve, továbbá a videokártyák általános felépítését is megismertük ebből a nézőpontból.

Miután Gustafson törvényével elméletben bebizonyítottuk, hogy az átültetésnek érdemes nekikezdeni, bemutatásra kerültek a GPU-ra optimalizált adattároló struktúrák és az átültetendő vessels modell részletes leírása is.

A kártyára portolás után megkezdődhetett a kapott teljesítmény részletekbe menő vizsgálata. Harmincöt hálózaton összesen négyszázhetven tesztet futtattunk, melyek megmutatták egyrészt, hogy minél nagyobb a hálózat, annál inkább megéri a videokártyán való futtatás, másrészt azt, hogy a különféleképpen felépített gráfokban, még ha összesen ugyanannyi csomópontjuk és élük is van, más sebességgel fut a szimuláció.

Összegezve, a Turbine CUDA-ra történő portolása rendkívül sikeres volt, hiszen a szimulált teszthálózatokon átlagosan 82-szeres gyorsulást sikerült elérni, ami kiemelkedőnek számít.

Irodalomjegyzék

[1] Silicon Graphics, Inc., NVIDIA Tesla GPU Computing, 2012, http://www.sgi.-com/pdfs/4325.pdf , utolsó hozzáférés: 2013. október 10.

[2] NVIDIA, CUDA C Programming Guide, 2013., http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf , utolsó hozzáférés: 2013. október 10.

[3] NVIDIA, History of GPU Computing, 2013., http://www.nvidia.com/object/cuda_home_new.html , utolsó hozzáférés: 2013. október 10.

[4] NVIDIA, GPU applications, 2013., http://www.nvidia.com/object/gpu-applications-domain.html , utolsó hozzáférés: 2013. október 12.

[5] Kathleen M. Carley, Dynamic Network Analysis, 2003

[6] Lada Adamic, Natalie Glance, The Political Blogosphere and the 2004 U.S. Election: Divided They Blog, 2005

[7] Anna Nagurney, Dynamic Networks: Recent Results and Applications, 2006., http://www.eecs.harvard.edu/~parkes/nagurney/nagurney.pdf , utolsó hozzáférés: 2013. október 16.

[8] Barabási Albert-László, Behálózva - A hálózatok csodálatos világa a sejtektől a világhálóig, 2005., http://mindentudas.hu/elodasok-cikkek/item/117-behálózva-a-hálózatok-csodálatos-világa-a-sejtektől-a-világhálóig.html , utolsó hozzáférés: 2013. október 25.

[9] Albert-Laszlo Barabasi, Reka Albert, Emergence of scaling in random networks, 1999.

[10] Szalay, K. Z. and Csermely, P., Perturbation centrality: a novel centrality measure obtained by the general network dynamics tool, Turbine, 2013.

[11] Réka Albert and Albert-László Barabási, Statistical mechanics of complex networks, 2002.

[12] Duncan J. Watts and Steven H. Strogatz, Collective dynamics of 'small-world' networks, 1998

[13] Erdős, Paul; A. Rényi, On the evolution of random graphs, 1960.

[14] NVIDIA, GTX 580 specifikáció, 2013., http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-580/specifications , utolsó hozzáférés: 2013. október 10.

[15] NVIDIA, Kepler GK110, , http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf , utolsó hozzáférés: 2013. október 10.

[16] Cliff Woolley, NVIDIA, CUDA Overview, 2011., http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/02-cuda-overview.pdf , utolsó hozzáférés: 2013. október 10.

[17] NVIDIA, CUDA Programming Overview, 2008, http://www.sdsc.edu/us/training/assets/docs/NVIDIA-02-BasicsOfCUDA.pdf , utolsó hozzáférés: 2013. október 10.

[18] American National Standards Institute, Inc., ANSI/IEEE 754-1985. American National Standard - IEEE Standard for Binary Floating-Point Arithmetic, 1985.

[19] American National Standards Institute, Inc., IEEE 754-2008, 2008.

[20] Nathan Whitehead, Alex Fit-Florea, Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs, 2011.

[21] NVIDIA, CUDA C Best Practices Guide, 2012, http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf , utolsó hozzáférés:

[22] John L. Gustafson, Reevaluating Amdahl's Law, 1988

[23] Tim C. Schroeder, Peer-to-Peer & Unified Virtual Addressing, , http://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf , utolsó hozzáférés: 2013. október 13.

[24] Valgrind Developers, Callgrind: a call-graph generating cache and branch prediction profiler, 2013., http://valgrind.org/docs/manual/cl-manual.html , utolsó hozzáférés: 2013. október 22.

Ábrajegyzék